Disseny i implementació d'eines didàctiques per facilitat l'aprenentatge de principis de la Física

Autor: Alex Anglarill Gobern

Universitat de Lleida
Escola Politècnica Superior

Directors: Josep Mª Ribó Balust
Miquel Carrera Vilanova

Enginyeria Tècnica en Informàtica de Gestió

Setembre 2007

1. Introducció i objectius

En les pàgines d'aquesta memòria s'hi troba l'inici d'un projecte de grans dimensions, l'objectiu del qual és crear un conjunt d'eines didàctiques per a millorar la comprensió i l'aprenentatge dels conceptes físics que complementi el temari impartit a les aules. El resultat d'aquest projecte inicial és crear un conjunt d'aplicacions com a complement al tema de l'electromagnetisme, on es pugui experimentar amb els coneixements obtinguts i aprendre mitjançant la pràctica. En un món com l'actual on accés a la informació és universal gràcies a Internet, la realització d'aquest projecte és per a una universitat, font de coneixement, quasi una necessitat.

Una bona manera d'entendre els conceptes impartits a les aules són els exemples o demostracions basats en la simulació del comportament d'un determinat sistema físic. En el camp de la informàtica un bon exemple seria una aplicació que permetés interactuar amb ella i que ensenyés i demostrés el concepte o principi físic en el qual està basat. En una pàgina *web* una de les formes d'aplicacions més manejables i utilitzades són els *applets*. Aquestes són aplicacions fetes en Java (Vegeu [13]) molt fàcilment incorporades a una pàgina *web* que permeten un alt grau d'interactivitat per part que l'usuari que vulgui experimentar amb ell.

La veritat és que a Internet ja existeixen un tipus d'*applets* dedicats a la Física anomenats Physlets (Vegeu [1] i [2]), matemàticament molt potents però molt limitats gràficament. Donat aquesta versatilitat matemàtica s'ha utilitzat per a crear un dels *applet* d'aquest projecte. Però donat que l'objectiu és que l'usuari que interactuï amb els *applets* no hagi de desxifrar una imatge sinó que fàcilment tingui l'entorn del concepte clar, vam optar per utilitzar una biblioteca amb més recursos gràfics, encara que matemàticament menys dotat, com és la biblioteca de Sun anomenada Java3D. (Vegeu [6],[7],[8],[9] i [10])

Per sintetitzar, els objectius d'aquest projecte serien:

- Analitzar les possibilitats de disseny i implementació d'aplicacions didàctiques basades en l'electromagnetisme a partir de la biblioteca dels Physlets. Aquest objectiu es concreta en l'estudi de la tecnologia Fislets (Physlets) i en la creació de l'*Applet* 1.
- Analitzar les possibilitats de disseny i implementació d'aplicacions didàctiques basades en l'electromagnetisme a partir de la biblioteca Java3D, posant especial èmfasi en les seves possibilitats gràfiques i de disseny 3D. Aquest objectiu es concreta en l'estudi de la tecnologia Java3D i en la creació de l'*Applet* 2.
- Donar l'opció als possibles usuaris d'aquestes aplicacions a poder experimentar-hi mitjançant la modificació de paràmetres interactius que modifiquin variables físiques del sistema electromagnètic en curs.

2. Anàlisi de tecnologies

2.1.Tecnologia: Fislets

Que són els Fislets?

Els Fislets, o Physlets (Vegeu [1] i [2]), són un conjunt d'*applets* Java que van ser creats per a l'ensenyança de la física al Davidson College, una escola d'educació superior de l'estat de Carolina del Nord (EEUU).
La finalitat d'aquests *applets* és la creació d'aplicacions que mitjançant una gran aposta per la interactivitat i facilitat d'ús, per a facilitar l'ensenyança de la física a les aules.

Els Applets

Existeix una gran quantitat d'*applets*, els quals intenten abastar la major quantitat d'entorns on es pot plasmar un determinat principi físic.

Alguns d'aquests són:

- *SApplet*: És la superclasse de tots els altres *applets*. Proporciona interconnectivitat en els *applets* de la biblioteca i operacions comunes i bàsiques. Entre les quals hi ha l'accés a l'animació del rellotge, a les dades de registre i dels "*listeners*", i la utilització del JavaScript en les connexions de dades.
- *Animator*: Serveix per fer animacions amb objectes geomètrics bàsics o amb imatges, definint la seva trajectòria o per l'efecte de forces.
- *BField:* Utilitzat per manipular camps i càrregues magnètiques. (Vegeu figura 2.1)

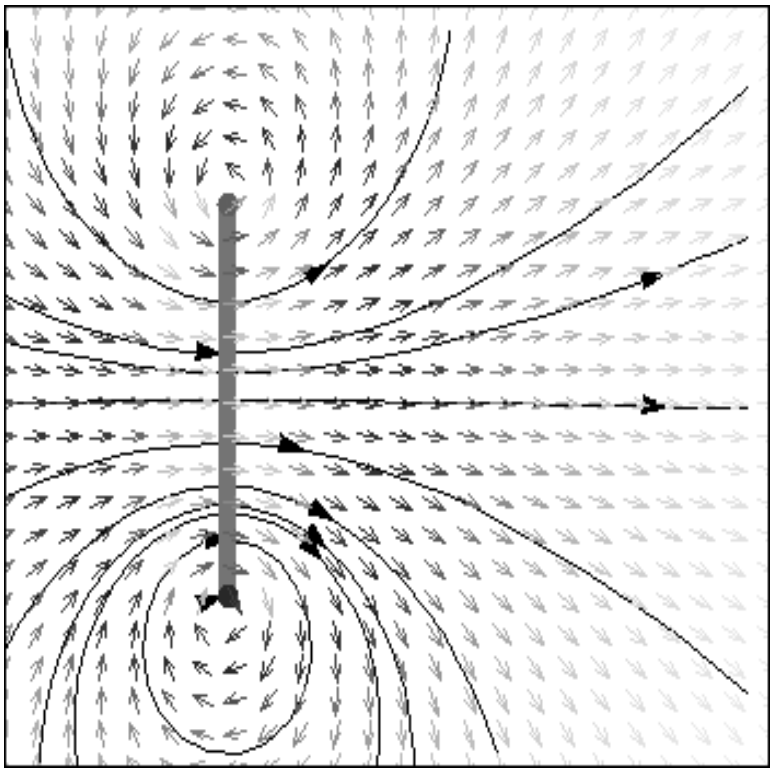

Figura 2.1. BField simulant un solenoide amb línies de camp

- *EField*: Utilitzat per manipular camps i càrregues elèctriques.
- Paquet *circuit*: Les classes que inclou són *IVApplet, IZApplet, LoadApplet, LRCApplet, i RCApplet*. Donen suport a problemes de circuits elèctrics. Donen la capacitat de crear circuits elèctrics de corrent continu i alterna, a més a més de poder fer càlculs amb ells. (Vegeu figura 2.2.)

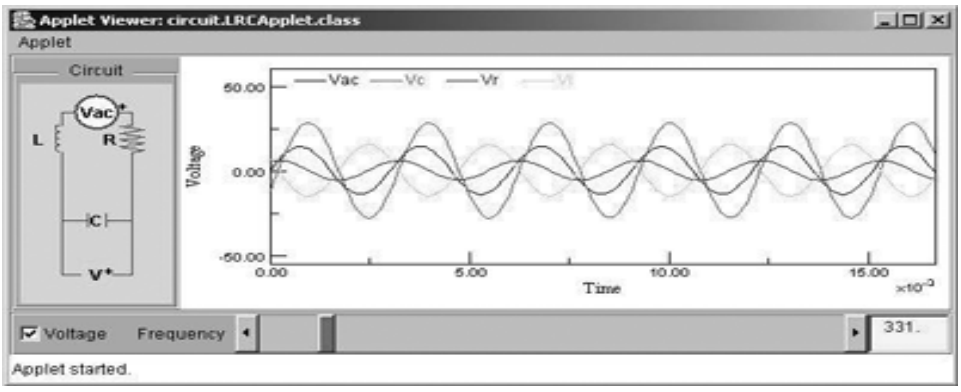

Figura 2.2. Simulació d'un circuit amb una resistència, un inductor i un condensador

- *DataTable*: Aquesta classe s'utilitza per a complement a una altra classe i serveix per mostrar dades en forma de taula.
- *Faraday*: Serveix per experimentar amb el model d'inducció magnètica d'una espira que llisca per uns eixos dintre d'un camp magnètic. (Vegeu figura 2.3.)

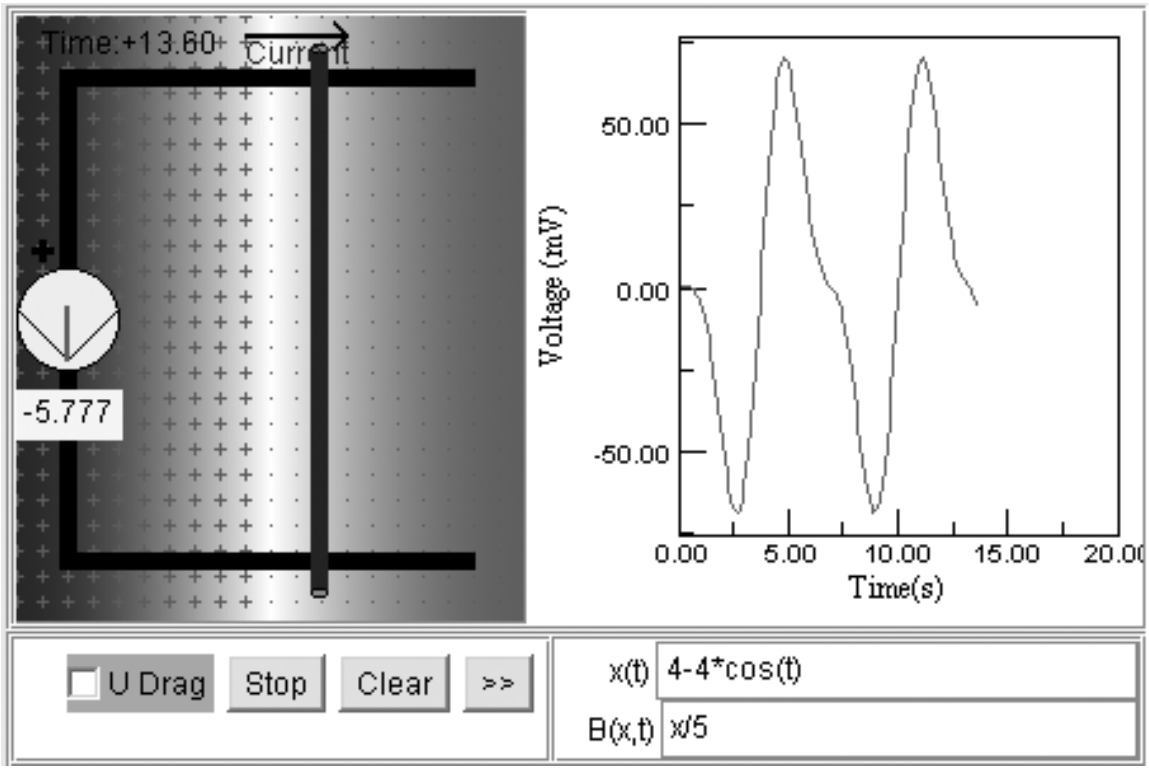

Figura 2.3. Exemple de la llei de Faraday i Lenz

- *OpticsApplet*: Serveix per experimentar amb elements òptics (lents, miralls, focus, etc). (Vegeu figura 2.4.)

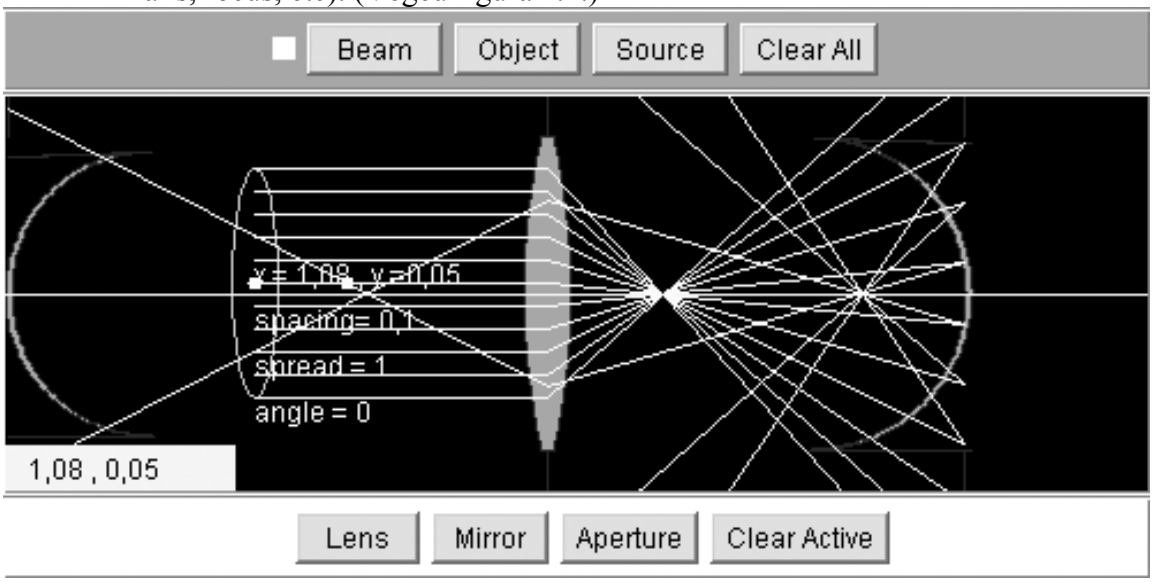

Figura 2.4. Exemple on apareixen diversos elements òptics

- *Stools4.jar*: Aquest *jar* s'inclou en totes les clàusules "*applet*". Es compon d'un gran nombre de classes utilitzades per al funcionament intern de l'*applet*, no accessibles per l'usuari.

Mètodes d'utilització

Existeixen dues formes d'utilitzar els *applets* d'aquesta biblioteca. Una primera que es compon d'una clàusula "*applet*" on s'inclou la crida (No tots els *applets* ho permeten). Una segona que a més a més de la clàusula "*applet*" inclou un codi JavaScript que interactua amb l'*applet*.
L'estructura d'una pàgina web seria:

<html>
<head>
<script LANGUAGE="JavaScript">
... Aquí van les funcions...
</script>
</head>
<body>
... Aquí va el codi *html* de la *web*...
<applet>
... Aquí és on es crida l'*applet* i s'inicialitzen els paràmetres que es necessiten...
</applet>
... Aquí va el codi *html* de la *web*...
</body>
</html>

Clàusula APPLET

Un exemple seria:

```
<applet code="animator4.Animator.class"
codebase="../../classes"
archive="Animator4_.jar,STools4.jar" name="Animator"
width="250" height="250" hspace="0" vspace="0"
align="Middle">
        <param name="FPS" value="10">
        <param name="ShowControls" value="false">
        <param name="dt" value="0.01">
        <param name="PixPerUnit" value="10">
        <param name="GridUnit" value="1.0">
</applet>
```

La clàusula *APPLET* porta incorporats alguns paràmetres que necessita com són:

- *Code*: indica la classe que ha d'executar que busca a partir dels arxius indicats en el paràmetre *archive*, per tant ha de contenir tota la ruta (en el cas que estigui dins d'un *package*).
- *Codebase*: ruta on ha de trobar els arxius del paràmetre *archive*.
- *Archive*: arxius que necessita per executar l'*applet*. Normalment són arxius *jar* i un d'ells és *STools4.jar*.

- *Name*: és el paràmetre que utilitzen els navegadors per referenciar els *applets*. Alguns navegadors utilitzen en el seu lloc el paràmetre *id*. Per evitar incompatibilitats és útil utilitzar-los tots dos, amb el mateix valor. Encara que la W3C vol substituir-los per *object*.
- *Width* i *height*: serveixen per definir la mida de l'*applet*.

A més a més dels paràmetres que defineix el llenguatge *html* (els anteriors). Aquests *applets* poden necessitar uns paràmetres que requerir per iniciar-se. La seva major o menor necessitat depèn de l'*applet* i de l'usuari que vulgui utilitzar-lo. Per utilitzar-los és necessari la clàusula <*param*> que va acompanyada de *name* per al nom del paràmetre i *value* per al valor que se li vulgui assignar. Alguns dels paràmetres són:

- *FPS*: són les imatges per segon que es calcularan i mostraran en l'*applet*. No s'ha de posar ni un valor excessivament baix o alt ja que produiria parpelleig o moviment a cops, respectivament. Un valor de 10 seria adequat.
- *ShowControls*: serveix per mostrar o amagar els controls per defecte de l'*applet*. Si els vol utilitzar uns de propis és útil amagar-los.
- *Dt*: serveix per definir la quantitat de temps gastat en cada imatge d'una animació. Per exemple si s'utilitza un *dt* de 0.1 i un *FPS* de 10 seria una animació en temps real.
- *Resources*: serveix per definir l'idioma del vocabulari de l'*applet*. Els arxius a utilitzar estan en el directori dels *applets* i tenen la forma *nomdeapplet_es.rc*, el text abans del punt fa referència a l'idioma (*es* espanyol, *en* anglès, etc.). Per exemple: *animator_es.rc*.

JavaScript

Javascript és un llenguatge interpretat, que no necessita compilar, utilitzat principalment en pàgines *web*, molt semblant en sintaxi a Java. Però a diferència d'aquest no està orientat a objectes sinó basat en prototipus que tracta principalment de la clonació de variables bàsiques que serveixen de prototipus.
En el nostre cas les funcions que es poden crear en l'apartat pertinent afectaran a l'*applet* i tenen accés a aquest.
La forma que té un *script* és la següent:

```
<script LANGUAGE="JavaScript">
function demo(){
  document.Animator.setAutoRefresh(false);
  document.Animator.setDefault();
 ...
}
```

</script>

En l'interior d'una clàusula *script* s'hi pot afegir tantes funcions com és vulgui.
Per accedir a l'*applet* cal utilitzar la següent sintaxi: *document.nomdeapplet.*, on *nomdeapplet* ha de ser igual al paràmetre *name/id* definit amb anterioritat, seguit del mètode al que es vol utilitzar del conjunt de funcions exposades a l'API de la biblioteca.

Com utilitzar les funcions

Per utilitzar les funcions es pot fer de dues maneres, utilitzant *hipervincles* o formularis.

Hipervincles: els *hipervincles* són normalment enllaços a pàgines *web* però també es poden utilitzar per enllaçar amb d'altres tipus d'arxius o cridar funcions de la mateixa *web* de la forma:
Començar

Formularis: els formularis són parts d'un codi *HTML* que va dins d'una clàusula *form*. Els formularis s'utilitzen per incloure una sèrie d'elements en una pàgina *web* com ara botons, quadres de text, etc. El disseny de formularis és molt fàcil si s'utilitza un editor de pàgines *web* però si no la seva sintaxi és molt senzilla:

```
<form name="nom">
        <input type="text" name="nomtext" size="10"
        value="0">
        <input type="button" value="marcha"
        onclick="funcio(document.nom.nomtext.value)">
        ...
</form>
```

Aquí hi ha un exemple on es crea un formulari de nom *nom*, el primer *input* crea un quadre de text de 10 caràcters de valor inicial 0 i de nom *nomtext*. El segon *input* és un botó on hi ha escrit *marcha* i que al prémer-lo es crida una funció anomenada *funcio* amb un paràmetre que és el valor del quadre de text. Com es pot veure en l'exemple, per accedir al valor del quadre de text cal utilitzar, a l'igual que a l'*applet*, mitjançant la sintaxi *document.nomformulari.nomtext.value*.

2.2. Tecnologia: Java3D

Que és Java3D?

Java3D és una extensió de Java2 que permet representar i interactuar amb gràfics en tres dimensions. L'API de Java3D està formada per una jerarquia de classes que permeten crear i manipular tota classe d'objectes i estructures, amb la finalitat d'obtenir visualitzacions, animacions i *renderitzacions* en 3D. (Vegeu [6],[7],[8],[9] i [10])

Tots els objectes es creen en un univers virtual, i posteriorment són *renderitzats*. El procés de *renderització* és un procediment de càlcul molt complex destinat a generar una imatge en 2D a partir d'una escena en 3D. A causa de l'automatització del *renderitzat*, tot programa de Java3D ha de situar els objectes (tant visuals com variables) utilitzats en una estructura en forma d'arbre anomenada *diagrama d'escena (també Scene Graph)*, que inclou els objectes creats i la manera com seran representats.

Diagrama d'escena

L'estructura d'un programa en Java3D té forma d'arbre. Cada objecte representat ve associat a dos objectes elementals que són la geometria i l'aparença.

Java3D porta definits varis objectes primitius: esfera, con, caixa i cilindre. Per a poder crear objectes més complexes cal recórrer als Grups de Transformació (*Transform Group*), que ens permeten situar objectes en l'espai i crear animacions. Tot Grup de Transformació ha d'estar enllaçat a un altre , però el primer de tots ells ha de ser un Grup de Branca (*BranchGroup*). Aquest està lligat al punt de referència (*Locale*) i aquest a la vegada a l'univers virtual (*VirtualUnivers*).

Els diagrames d'escena utilitzen una simbologia per representar cada objecte. (Vegeu figura 2.5.)

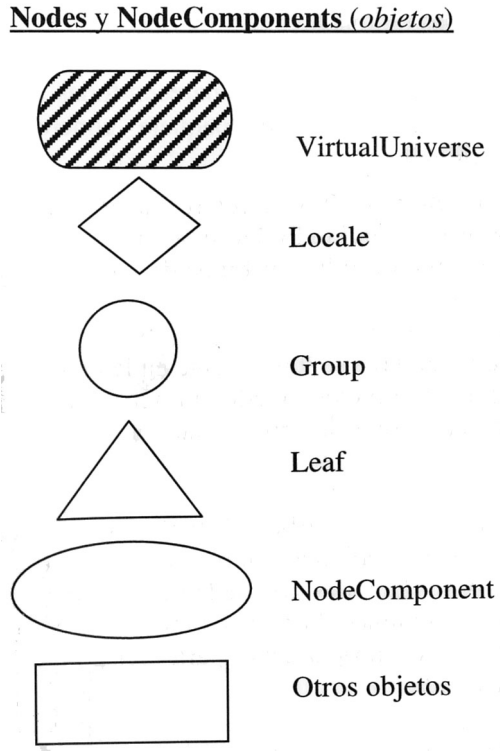

Figura 2.5. Simbologia dels diagrames d'escena

Per enllaçar aquests objectes s'utilitzen dos tipus de fletxes:

——→ Relació de dependència pare-fill (*parent-child*).

- - - - - - - → Referència

La relació pare-fill té la restricció que cada node (classe) de *Group* (grup d'objectes) del diagrama pot tenir molts fills, però només un pare, i cada node de *Leaf* (fulla o objecte final) solament pot tenir un pare i cap fill.

L'arbre tampoc pot tenir cicles i solament hi pot haver un camí entre el node arrel i cada una de les fulles (*Leaf*).

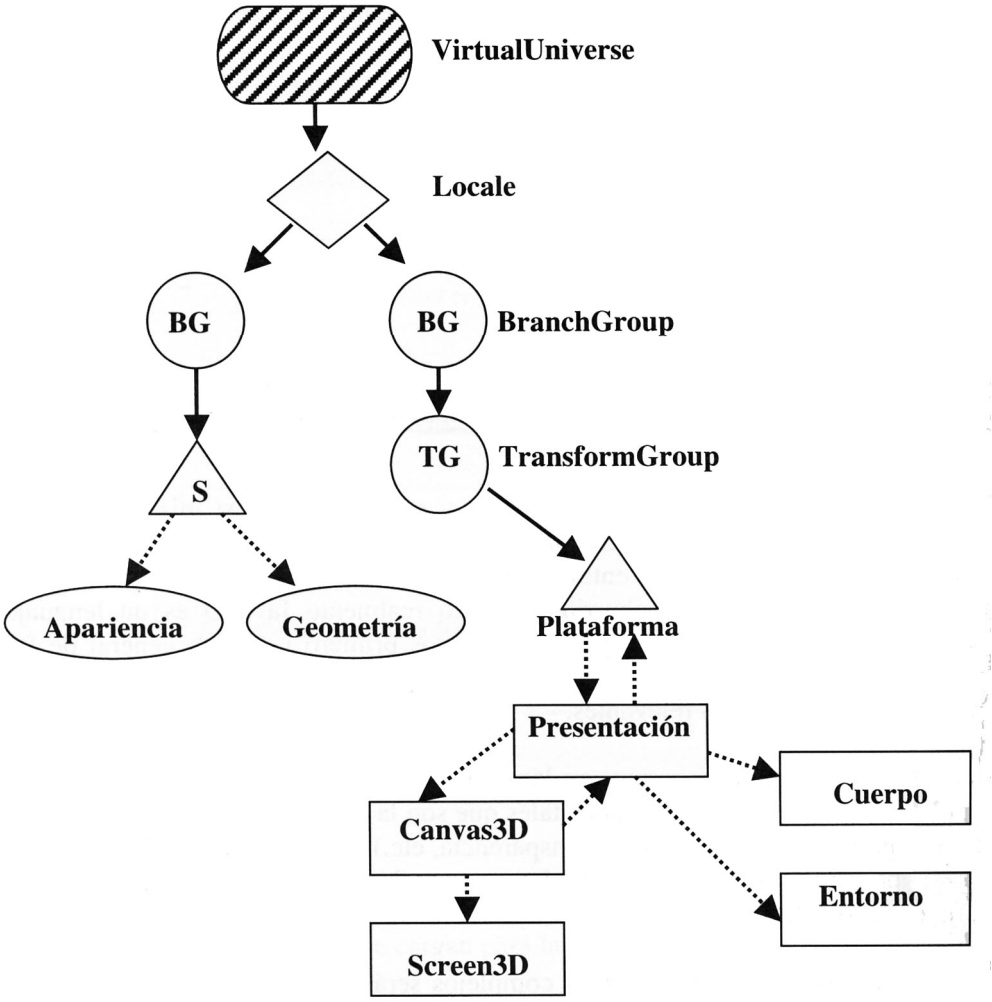

Figura 2.6. Exemple de diagrama d'escena

A la imatge (Vegeu figura 2.6.) es pot observar un exemple de diagrama d'escena. A dalt de tot hi ha un univers virtual (*VirtualUnivers*) amb un únic punt de referència (*Locale*) del que en surten dues rames, la de Contingut i la de Representació. La primera genera un objecte 3D (S de *Shape*) amb l'aparença i Geometria associades; i la segona rama defineix l'entorn de representació per pantalla.

Jerarquia de classes

Com a l'API de Java2 les classes de Java3D estan organitzades de major a menor importància en nivells de jerarquia. (Vegeu figura 2.7.)

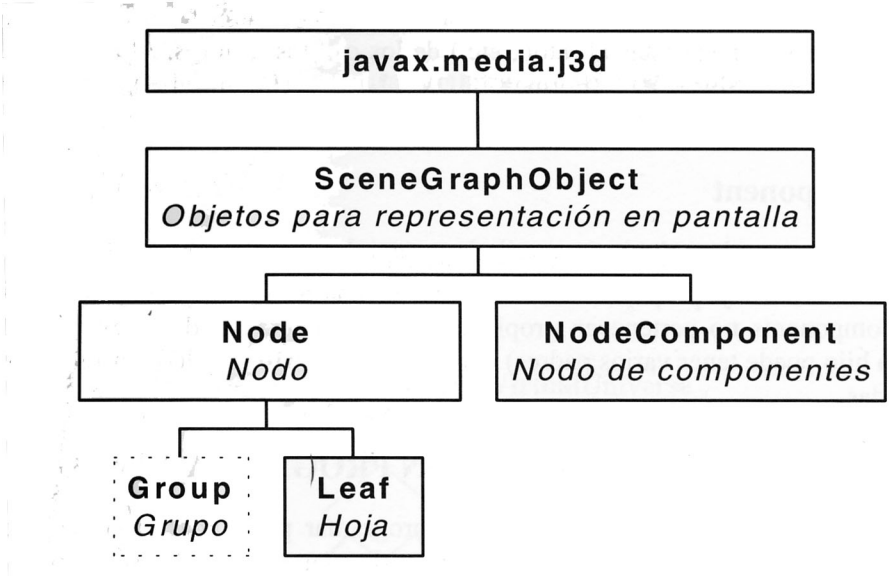

Figura 2.7. Jerarquia de classes en Java3D

- Node: És la superclasse abstracta de les classes *Group* i *Leaf*, defineix les subclasses per representar diagrames d'escena.
- *Group*: És la superclasse abstracta utilitzada per definir transformacions de posició i d'orientació sobre els objectes visuals. Dos de les seves subclasses són: *BranchGroup (BG)* i *TransformGroup (TG)*.
- *Leaf*: És la superclasse abstracta utilitzada per definir formes, sons i comportaments. Algunes de les seves subclasses: *Shape3D*(formes 3D), *Light* (il·luminació), *Behavior* (comportament) i *Sound* (sons).
- *NodeComponent*: És la superclasse abstracta utilitzada per definir les característiques d'una classe *Shape3D*, com ara la seva geometria, l'aparença i propietats del material del que està fet. Donat que una mateixa classe *NodeComponent* pot estar lligada a diversos objectes, en els diagrames d'escena estan lligats als objectes per referències.

Un programa de Java3D

En tot programa en Java3D s'han de seguir una sèrie de passos o etapes que posteriorment seran enllaçades:

1. Crear l'objecte base: *Canvas3D*.
2. Crear l'objecte univers virtual: *VirtualUnivers*.
3. Crear el punt referència : *Locale*.
4. Construir la Rama de Representació.
    a. Crear l'objecte Presentació.

   b. Crear l'objecte Plataforma.
   c. Crear el Cos i l'Entorn.
5. Construir la Rama de Contingut.
6. Compilar el diagrama d'escena.
7. Inserir altres rames dins del punt de referència.

Però en la majoria de programes no cal fer tots els passos, perquè amb la utilització d'una classe anomenada *SimpleUnivers* (Vegeu figura 2.8) que conté tota la informació de la Rama de Representació, s'estalvia una part que és gairebé comú a tots els programes Java3D.

Els passos que s'han de prendre utilitzant *SimpleUnivers* són:

1. Adaptar l'objecte *SimpleUnivers*.
2. Crear l'objecte base: *Canvas3D*.
3. Construir la Rama de Contingut.
4. Compilar el diagrama d'escena.
5. Inserir la Rama de Contingut dins del punt de referència (*Locale*) de l'objecte *SimpleUnivers*.

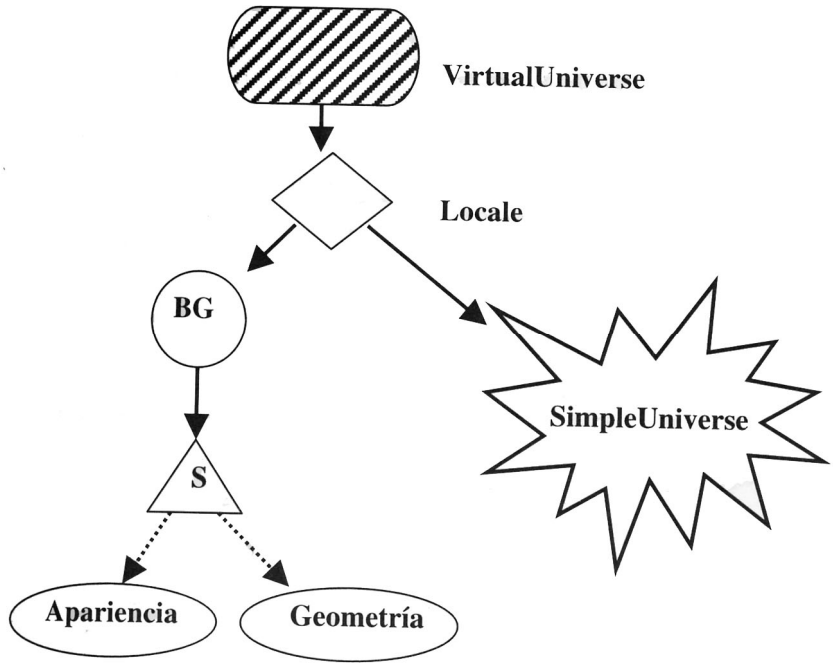

Figura 2.8. Exemple de la utilització de *SimpleUnivers*

La classe *SimpleUnivers* té varis constructors, però els més utilitzats són:

- El constructor buit. *SimpleUnivers()*

  Crea el punt de Referència (*Locale*) i la Rama de Presentació.

- *SimpleUnivers(Canvas3D* canvas3d*)*

  Crea el punt de Referència (*Locale*) i la Rama de Presentació, hi associa com a Suport o base un objecte *Canvas3D*.

Punt de vista inicial de la classe *SimpleUnivers*:

Per representar un univers en 3D en una pantalla 2D es projecten els objectes en un pla imaginari (la pantalla). Per defecte, el punt de vista inicial és l'origen de coordenades (0,0,0). Com que els objectes, a falta de presentar les transformacions, inicialment es representen en l'origen de coordenades, des del punt de vista inicial no veurem cap objecte. Per a solucionar aquest contratemps *ViewingPlatform* proporciona un mètode, *setNominalViewingTransform()*, que situa el punt de vista a (0,0,2.41), que significa a 2.41 de la pantalla, distància que permet observar els objectes. L'objecte *ViewingPlatform* de l'univers actual s'obté mitjançant el mètode *SimpleUnivers.getViewingPlatform()*.

Un exemple de programa senzill podria ser:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.awt.GraphicsConfiguration;

import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.geometry.*;
import javax.media.j3d.Canvas3D;
import javax.media.j3d.BranchGroup;


public class PrimerDibujo3D extends Applet{

   public PrimerDibujo3D(){

                setLayout(new BorderLayout());
                GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
                Canvas3D canvas3D = new Canvas3D(config);
                add("Center", canvas3D);

                BranchGroup scene = createSceneGraph();

                scene.compile();

                SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
                simpleU.getViewingPlatform().setNominalViewingTransform();
                simpleU.addBranchGraph(scene);

   }

   public BranchGroup createSceneGraph() {

                BranchGroup objRoot = new BranchGroup();
                objRoot.addChild(new ColorCube(0.4);
                return objRoot;
   }

   public static void main(String[] args) {
                Frame frame = new MainFrame(new PrimerDibujo3D(), 256, 256);
   }

}
```

Capacitats

Un tema que afecta a gairebé totes les classes de la biblioteca és el de les capacitats (*capabilities*). Aquestes són propietats que poden activar-se i donen accés a l'objecte a l'hora de llegir o modificar el seu contingut en temps d'execució. Cada classe té les seves pròpies capacitats i les que ha heretat de la seva superclasse.

Per a tractar aquestes capacitats la classe que en posseeix proporciona els següents mètodes:

- *void clearCapability(int* bit*)*: Esborra (desactiva) la capacitat identificada pel bit.
- *boolean getCapability(int* bit*)*: Recupera l'estat (si està activada) de la capacitat identificada pel bit.
- *void setCapability(int* bit*)*: Activa la capacitat identificada pel bit.

Alguns exemples de capacitats i algunes de les utilitzades:

- *TransformGroup*:
  - ALLOW_TRANSFORM_READ: Permet a l'objecte al qual s'aplica donar informació dels objectes lligats a ell (p.e. *Transform3D*).
  - ALLOW_TRANSFORM_WRITE: Permet que l'objecte al qual s'aplica sigui modificat.
- *Group*:
  - ALLOW_CHILDREN_EXTEND: Per afegir un objecte fill en una branca ja compilada.
  - ALLOW_CHILDREN_READ: Permet accedir als fills d'una branca ja compilada.
  - ALLOW_CHILDREN_WRITE: Permet accedir i modificar els fills d'una branca ja compilada.

Un exemple seria:

```
...
TransformGroup objSpin = new TransformGroup();
objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

...
```

Classes primitives

Abans d'exposar els temes més importants de la biblioteca, cal destacar una sèrie de grups de classes essencials que s'utilitzen en gairebé tots els aspectes de la biblioteca. Aquests grups inclouen les classes per a definir transformacions, classes matemàtiques i classes geomètriques primitives.

Transformacions

- *Transform3D*: Els objectes d'aquesta classe defineixen transformacions de rotació, translació, compressió i expansió. Els objectes d'aquesta classe es poden combinar entre ells i així esdevenir objectes que inclouen varis tipus de transformació. Internament els objectes d'aquesta classe es representen per matrius 4 x 4 (3 coordenades més el temps).

  Els constructors que inclou aquesta classe permeten construir aquesta "matriu" de diferents formes i utilitzant diferents paràmetres. Entre els mètodes que inclou cal destacar els següents:

  - *void mul(Transform3D* t1*)*: s'utilitza per multiplicar l'objecte (*this*) per t1.
  - *void rotX(double* angle*)*: defineix una rotació sobre l'eix X en un cert nombre de radians "angle". Per a definir l'angle s'utilitza *Math.PI*. El mateix per *rotY* i *rotZ*.
  - *void set(Vector3f* translate*)*: defineix la translació amb el vector translate que serà sumat a l'objecte. La classe ofereix varis mètodes set per a diferents tipus de paràmetres.

  Aquesta classe no s'utilitza directament en el diagrama d'escena, sinó que s'aplica sobre un objecte *TransformGroup*, mitjançant el mètode *setTransform(Transform3D* transf*)* o el mateix constructor de la classe.

- *Vector3f*: Una altra classe que s'empra amb certa freqüència que serveix per definir un vector en un espai 3D amb variables decimals. Tampoc s'aplica directament, sinó sobre un objecte *Transform3D*. La biblioteca també inclou diverses classes semblants a aquesta per emmagatzemar un vector en diferents variables com és *Vector3d* (double) o diferents tipus de vectors (de 2 o 4 variables).

  Els constructors que inclou la classe són varis, tots per a la construcció d'un vector a partir de diferents entrades o classes. Alguns exemples:

  - *Vector3f()*: Construeix i inicialitza el vector neutre (0,0,0).
  - *Vector3f(float* x*, float* y*, float* z*)*: Construeix i inicialitza el vector amb els components introduïts (x,y,z).

Un exemple de *Transform3D*:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class SegundaAnimacion extends Applet {

   public BranchGroup createSceneGraph() {

                BranchGroup objRoot = new BranchGroup();
                Transform3D rotate = new Transform3D();
                Transform3D tempRotate = new Transform3D();
                Transform3D tempRotate2 = new Transform3D();

                rotate.rotZ(Math.PI/4.0d);
                tempRotate.rotY(Math.PI/5.0d);
                tempRotate2.rotX(Math.PI/5.0d);
                tempRotate2.mul(tempRotate);
                rotate.mul(tempRotate2);

                TransformGroup objRotate = new TransformGroup(rotate);

                TransformGroup objSpin = new TransformGroup();
                objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

                ...
```

Classes matemàtiques

Java3D defineix una sèrie de classes abstractes de propòsit matemàtic les subclasses de les quals tenen un ús generalitzat en tot programa de la biblioteca. Aquestes classes abstractes són: *Tuple2f, Tuple3f, Tuple4f* (juntament amb les seves variants de precisió *double*, *byte*, *int*). Aquestes classes posen les bases per al tractament de *tuples* de 2, 3 ó 4 números, necessari en un sistema de coordenades com és el 3D.

Algunes de les seves subclasses (juntament amb les variants en precisió) són:

- *Point3f*: utilitzada per definir punts en l'espai.
- *Color3f*: utilitzada per definir colors mitjançant el valor dels tres colors primaris.
- *Vector3f*: anomenada amb anterioritat, serveix per definir vectors i desplaçaments en l'espai.
- *TexCoord3f*: utilitzada per situar textures.
- *Quad4f*: utilitzada per definir quaternions, tipus de variable d'aplicació en transformacions.

Classes geomètriques primitives

Java3d proporciona un conjunt de classes que representen formes primitives d'objectes senzills:

Shape3D: Aquesta classe és una subclasse de *Leaf* (fulla), el que significa que els objectes d'aquesta classe no tindran fills (Nodes). La classe per si sola no defineix cap aspecte de l'objecte 3D, per a definir-lo cal referenciar-li dos *NodeComponent*: *Geometry* (geometria) i *Appearance* (aparença).

Alguns constructors de la classe són:

- *Shape3D()*: Construeix i inicialitza un objecte sense geometria i aparença.
- *Shape3D(Geometry* geo*, Appearance* app*)*: Construeix i inicialitza un objecte *Shape3D* amb els paràmetres entrats.

Aquesta classe proporciona mètodes get/set per a l'aparença i la geometria. I també conté importants capacitats que proporcionen la possibilitat de canviar la geometria i l'aparença:

- ALLOW_GEOMETRY_WRITE, ALLOW_GEOMETRY_READ
- ALLOW_APPEARANCE_WRITE, ALLOW_APPEARANCE_READ

Primitives: en el paquet *com.sun.j3d.utils.geometry.Primitive,* Java3D ens proporciona un conjunt de classes de figures geomètriques senzilles: *Box* (caixa), *Cone* (con), *Cylinder* (cilindre) i *Sphere* (esfera). El seu ús és bastant senzill i

tenen una funcionalitat limitada, reduïda a definir la seva aparença o a extreure a *Shape3D* alguna de les seves cares (definides per constants).

Text en 3D

Java3D proveeix dues formes d'introduir text en l'espai virtual. Aquestes són les que proporcionen les classes *Text2D* i *Text3D*.

La primera d'elles no presenta dificultat ja que només cal definir els paràmetres que necessita (text, color, font, mida i estil) i enllaçar-lo com a fill d'una de les rames de la Rama de Contingut donat que és una subclasse de *Shape3D*.

La utilització de *Text3D* presenta una mica més de complexitat, però seguint una sèrie de passos senzills se soluciona amb celeritat.

Els passos són:

- Crear un nou objecte *Font3D*. El seu constructor precisa d'un objecte Font (de Java2, *java.awt.Font*) i el grau d'extrusió en un objecte *FontExtrusion*.
- Amb l'objecte *Font3D*, una posició *Point3f* i un objecte *String* del que es vulgui escriure es crea un objecte *Text3D*.
- Aquest objecte *Text3D* (subclasse de *Geometry*) s'ha de definir com a geometria d'un objecte *Shape3D* i ja es pot utilitzar penjant-lo d'una branca de la Rama de Contingut.

Dos exemples, un de *Text2D* i un altre de *Text3D*:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Font;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.Text2D;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;


public class Text2D_I extends Applet {

    public Text2D_I() {
            setLayout(new BorderLayout());
            GraphicsConfiguration config =
            SimpleUniverse.getPreferredConfiguration();
```

```
                Canvas3D canvas3D = new Canvas3D(config);
                canvas3D.setStereoEnable(false);
                add(canvas3D);
                BranchGroup scene = createSceneGraph();
                SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
                simpleU.getViewingPlatform().setNominalViewingTransform();
                simpleU.addBranchGraph(scene);
        }

        public BranchGroup createSceneGraph() {
                BranchGroup objRoot = new BranchGroup();

                Text2D text2d = new Text2D("Text2D",new Color3f(0.0f, 1.0f, 1.0f), "Helvetica", 32,
                Font.ITALIC);

                objRoot.addChild(text2d);

                return objRoot;
        }

        public static void main(String[] args) {
                Frame frame = new MainFrame(new Text2D_I(), 256, 50);
        }
}


import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Font;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.Text2D;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;


public class Text3D_I extends Applet {

        public Text3D_I() {
                setLayout(new BorderLayout());
                GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();

                Canvas3D canvas3D = new Canvas3D(config);
                canvas3D.setStereoEnable(false);
                add(canvas3D);
                BranchGroup scene = createSceneGraph();
                SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
                simpleU.getViewingPlatform().setNominalViewingTransform();
                simpleU.addBranchGraph(scene);
        }

        public BranchGroup createSceneGraph() {

                BranchGroup objRoot = new BranchGroup();

                Transform3D translation = new Transform3D();
                translation.setTranslation(new Vector3f(0.0f, 0.0f, -4.0f));
                TransformGroup objMove = new TransformGroup(translation);
                objRoot.addChild(objMove);

                TransformGroup objSpin = new TransformGroup();
                objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
                objMove.addChild(objSpin);

                Appearance app = new Appearance();
                app.setMaterial(new Material());

                Font3D font3D = new Font3D(new Font("Times", Font.PLAIN, 1), new FontExtrusion());

                Text3D text = new Text3D(font3D, new String("Text3D"));

                text.setAlignment(Text3D.ALIGN_CENTER);
```

```
Shape3D textShape = new Shape3D();
textShape.setGeometry(text);
textShape.setAppearance(app);
objSpin.addChild(textShape);
...
```

Geometria

Evidentment amb les classes primitives exposades no es poden crear objectes complexes. Per aquest fi Java3D proporciona una sèrie de classes encapçalades per *Geometry* que engloben totes les formes (amb l'excepció dels *Loaders*) de crear formes geomètriques en l'univers virtual de Java3D. (Vegeu figura 2.9.)

Les subclasses de *Geometry* es divideixen en un gran grup de classes encapçalades per *GeometryArray* i en tres classes d'una temàtica diferent: *Text3D* (ja exposada), *Raster* i *CompressedGeometry*.

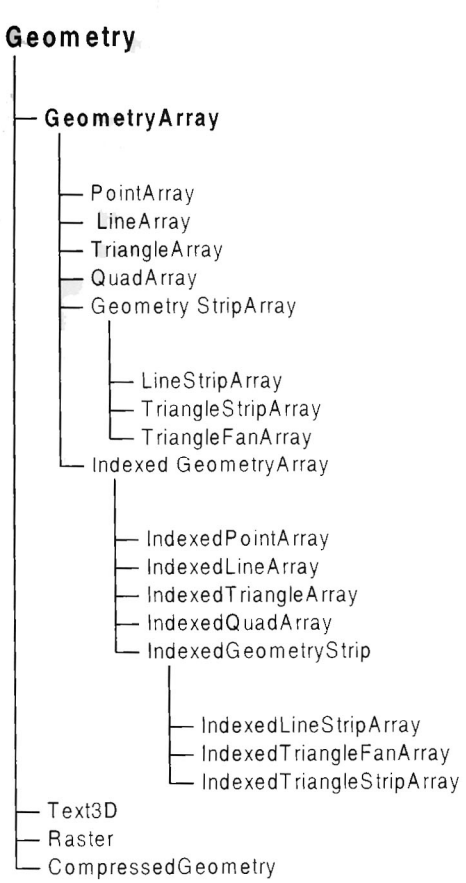

Figura 2.9. Subclasses de *Geometry*

GeometryArray

Abans d'exposar els seus subgrups cal destacar algunes de les constants necessàries per la utilització de totes les seves subclasses, emprades per a definir el format dels vèrtexs:

- COORDINATES: La utilització d'aquesta constant és obligatòria. Indica que el vèrtex emmagatzemarà les coordenades.
- NORMALS: El vèrtex emmagatzema el seu vector Normal. Útil si es volen utilitzar efectes lumínics.
- COLOR_3: Els vèrtexs seran del color que s'especifiqui sense transparència.
- COLOR_4: Els vèrtexs seran del color i la transparència que s'especifiqui.
- TEXTURE_COORDINATE_2: El vèrtex emmagatzema a quin vèrtex de la textura correspon, en una coordenada de 2D.
- TEXTURE_COORDINATE_3: El vèrtex emmagatzema a quin vèrtex de la textura correspon, en una coordenada de 3D.

El primer dels subgrups de *GeometryArray* comprèn les classes: *PointArray*, *LineArray*, *TriangleArray* i *QuadArray*. Aquestes classes serveixen, respectivament, per definir grups de punts, línies, triangles i quadrats.

Un exemple d'aquestes primeres classes:

```
import java.applet.Applet;
import java.awt.Frame;
import java.awt.BorderLayout;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.applet.MainFrame;
import javax.media.j3d.Canvas3D;
import javax.media.j3d.BranchGroup;
import javax.media.j3d.PointArray;
import javax.media.j3d.LineArray;
import javax.media.j3d.TriangleArray;
import javax.media.j3d.QuadArray;
import javax.media.j3d.Shape3D;
import javax.media.j3d.TransformGroup;
import javax.media.j3d.Transform3D;
import javax.media.j3d.Appearance;
import javax.media.j3d.PolygonAttributes;
import javax.vecmath.Point3f;
import javax.vecmath.Color3f;
import javax.vecmath.Vector3f;

public class GeomCadenas extends Applet{

        GeomCadenas(){

                this.setLayout(new BorderLayout( ));
```

```
        Canvas3D canvas3D=new Canvas3D(SimpleUniverse.getPreferredConfiguration());

        this.add(BorderLayout.CENTER, canvas3D);

        SimpleUniverse simpleU=new SimpleUniverse(canvas3D);
        BranchGroup scene=createScene();
        simpleU.addBranchGraph(scene);
        simpleU.getViewingPlatform().setNominalViewingTransform();

        this.add("Center",canvas3D);
}

BranchGroup createScene()
{
        BranchGroup scene=new BranchGroup();

        //Eje de las X
        LineArray ejeX=new LineArray(2,LineArray.COORDINATES|LineArray.COLOR_3);
        ejeX.setCoordinate(0,new Point3f(-1f,0f,0f));
        ejeX.setCoordinate(1,new Point3f(1f,0f,0f));
        ejeX.setColor(0,new Color3f(1f,0f,0f));
        ejeX.setColor(1,new Color3f(0f,0f,1f));

        //Eje de las Y
        LineArray ejeY=new LineArray(2,LineArray.COORDINATES|LineArray.COLOR_3);
        ejeY.setCoordinate(0,new Point3f(0f,-1f,0f));
        ejeY.setCoordinate(1,new Point3f(0f,1f,0f));
        ejeY.setColor(0,new Color3f(0f,1f,0f));
        ejeY.setColor(1,new Color3f(1f,0f,0f));

        //Cadena de Puntos
        PointArray punto=new PointArray(3,PointArray.COORDINATES|PointArray.COLOR_3);
        punto.setCoordinate(0,new Point3f(-0.8f,0.2f,0f));
        punto.setCoordinate(1,new Point3f(-0.2f,0.2f,0f));
        punto.setCoordinate(2,new Point3f(-0.5f,0.8f,0f));
        punto.setColor(0,new Color3f(1f,1f,0f));
        punto.setColor(1,new Color3f(1f,1f,0f));
        punto.setColor(2,new Color3f(0f,1f,1f));

        //Cadena de Líneas
        LineArray line=new LineArray(4,LineArray.COORDINATES|LineArray.COLOR_3);
        line.setCoordinate(0,new Point3f(-0.8f,-0.2f,0f));
        line.setCoordinate(1,new Point3f(-0.2f,-0.2f,0f));
        line.setCoordinate(2,new Point3f(-0.5f,-0.8f,0f));
        line.setCoordinate(3,new Point3f(-0.6f,-0.5f,0f));
        line.setColor(0,new Color3f(1f,0f,0f));
        line.setColor(1,new Color3f(0f,1f,0f));
        line.setColor(2,new Color3f(0f,0f,1f));
        line.setColor(3,new Color3f(1f,0f,1f));

        //Cadena de Triángulos
        TriangleArray triangle=new
        TriangleArray(6,TriangleArray.COORDINATES|TriangleArray.COLOR_3);
        triangle.setCoordinate(0,new Point3f(0.2f,0.1f,0f));
        triangle.setCoordinate(1,new Point3f(0.6f,0.1f,0f));
        triangle.setCoordinate(2,new Point3f(0.6f,0.5f,0f));
        triangle.setCoordinate(3,new Point3f(0.2f,0.6f,0f));
        triangle.setCoordinate(4,new Point3f(0.6f,0.8f,0f));
        triangle.setCoordinate(5,new Point3f(0.8f,0.8f,0f));
        for(int i=0;i<3;i++) triangle.setColor(i,new Color3f(1f,1f,0f));
        for(int i=4;i<6;i++) triangle.setColor(i,new Color3f(0f,1f,1f));

        //Cadena de Cuadrados
        QuadArray quad=new QuadArray(4,QuadArray.COORDINATES|QuadArray.COLOR_3);
        quad.setCoordinate(0,new Point3f(0.2f,-0.2f,0f));
        quad.setCoordinate(1,new Point3f(0.2f,-0.8f,0f));
        quad.setCoordinate(2,new Point3f(0.6f,-0.8f,0f));
        quad.setCoordinate(3,new Point3f(0.7f,-0.1f,0f));
        quad.setColor(0,new Color3f(1f,0f,0f));
        quad.setColor(1,new Color3f(0f,1f,0f));
        quad.setColor(2,new Color3f(0f,1f,1f));
        quad.setColor(3,new Color3f(1f,1f,0f));

        scene.addChild(new Shape3D(ejeX));
        scene.addChild(new Shape3D(ejeY));
```

```
            scene.addChild(new Shape3D(punto));
            scene.addChild(new Shape3D(line,cadenaApp()));
            scene.addChild(new Shape3D(triangle,cadenaApp()));
            scene.addChild(new Shape3D(quad,cadenaApp()));

            scene.compile();
            return scene;
    }
    ...
```

El segon subgrup correspon a les subclasses de *GeometryStrip* on la principal diferència amb el grup anterior és l'encadenament dels objectes:

- *LineStripArray*: Els objectes (línies) s'uneixen segons l'ordre dels vèrtexs. El principal constructor és *LineStripArray(int* vextexCount*, int* vextexFormat*, int* stripVertexCounts[]*)*.
- *TriangleStripArray*: Els objectes (triangles) s'encadenen mitjançant la norma, vàlida a partir del 4rt vèrtex inclòs, que el nou vèrtex s'enllaçarà amb el dos anteriors.
- *TriangleFanArray*: Els objectes (triangles) s'encadenen mitjançant la norma que cada nou vèrtex s'enllaçarà amb el primer i l'últim vèrtex.

Aquestes tenen en comú que utilitzen un paràmetre, anomenat **stripVertexCounts**[], que s'utilitza per definir el nombre de grups amb que es volen dividir el total de vèrtexs i el nombre de vèrtexs que hi haurà a cada grup.

Un exemple:

```
import java.applet.*;
import java.awt.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class GeomStrip extends Applet {

  public GeomStrip() {

        GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
        Canvas3D canvas = new Canvas3D(config);
        this.setLayout(new BorderLayout());
        this.add(canvas, BorderLayout.CENTER);

        SimpleUniverse universe = new SimpleUniverse(canvas);
        universe.getViewingPlatform().setNominalViewingTransform();

        BranchGroup scene = createSceneGraph();

        universe.addBranchGraph(scene);
  }
```

```
 private BranchGroup createSceneGraph() {
        BranchGroup objRoot = new BranchGroup();

        int n=0;
        int totalN=16;
        Point3f[] coords = new Point3f[totalN];

        int[] stripVertexCounts = {6,10}; //Divideix els vèrtexs en dos grups de 6 i 10

        for(n= 0; n < totalN;  n=n+2 ){
          coords[n] = new Point3f((float)(0.1*n-0.5), 0.5f, 0f);
          coords[n+1] = new Point3f((float)(0.1*n-0.8), -0.5f, 0f);
         }

        TriangleStripArray geometry = new
        TriangleStripArray(coords.length, GeometryArray.COORDINATES, stripVertexCounts);
        geometry.setCoordinates(0, coords);

        Shape3D shape = new Shape3D(geometry,app());

        objRoot.addChild(shape);

        return objRoot;
}
private Appearance app() {

      Appearance app=new Appearance();
      PolygonAttributes polyAtt=new PolygonAttributes();
      polyAtt.setPolygonMode(PolygonAttributes.POLYGON_LINE);
      app.setPolygonAttributes(polyAtt);

      return app;
}

 public static void main(String[] args) {
   GeomStrip applet = new GeomStrip();
   MainFrame frame = new MainFrame(applet, 350, 350);
 }
}
```

El tercer grup el formen les classes de geometria indexada (*IndexedGeometryArray*), molt útil si es volen repetir vèrtexs en el procés de formació de l'objecte geomètric. Com en el primer grup aquest està format per quatre classes que segueixen l'ordre dels vèrtexs:

- Punts: *IndexedPointArray*.
- Línies: *IndexedLineArray*
- Triangles: *IndexedTriangleArray*
- Quadrats: *IndexedQuadArray*

Aquestes classes segueixen el tipus de constructor de la seva superclasse:

- *IndexedGeometryArray(int* vertexCount*, int* vertexFormat*, int* indexCount*)*: El paràmetre **indexCount** indica el nombre de vèrtexs incloent les repeticions.

Per a poder concretar l'ordre dels índexs i en concret les seves repeticions, *IndexedGeometryArray* proporciona una col·lecció de mètodes, el més important dels quals és:

- *void setCoordinateIndices(int* index*, int* coordinateIndices[]*)*: On ***index*** indica el primer vèrtex, i ***coordinateIndices*** és l'ordre que seguirà a partir d'aquest.

Una segona part dins la geometria indexada, és una sèrie de classes, agrupades sota una d'abstracta *IndexedGeometryStripArray*. Aquesta sèrie és molt semblant a les anteriors classes de geometria indexada, però, com entre el primer i segon grup de *GeometryArray*, en els seus constructors s'inclou el paràmetre ***stripVertexCount*** que s'utilitza per dividir els vèrtexs en grups i formar diferents objectes.

Un exemple de geometria indexada que dibuixa un cub:

```
import java.applet.*;
import java.awt.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class IndexedGeom extends Applet {

  public IndexedGeom() {

    GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas = new Canvas3D(config);
    this.setLayout(new BorderLayout());
    this.add(canvas, BorderLayout.CENTER);

    SimpleUniverse simpleU = new SimpleUniverse(canvas);
    simpleU.getViewingPlatform().setNominalViewingTransform();

    BranchGroup scene = createSceneGraph();

    simpleU.addBranchGraph(scene);
  }

  private BranchGroup createSceneGraph() {
    BranchGroup root = new BranchGroup();

    Point3d[] coords = new Point3d[8];

    coords[0] = new Point3d(-0.6, 0.6, 0.6);
    coords[1] = new Point3d(-0.6, -0.6, 0.6);
    coords[2] = new Point3d(0.6, -0.6, 0.6);
    coords[3] = new Point3d(0.6, 0.6, 0.6);
    coords[4] = new Point3d(-0.6, 0.6, -0.6);
    coords[5] = new Point3d(-0.6, -0.6, -0.6);
    coords[6] = new Point3d(0.6, -0.6, -0.6);
    coords[7] = new Point3d(0.6, 0.6, -0.6);

    int[] indices = { 0, 1, 1, 2, 2, 3, 3, 0, 0, 4, 1, 5, 2, 6, 3, 7, 4, 5, 5, 6, 6, 7, 7, 4 };
```

```
IndexedLineArray geometry = new
IndexedLineArray(coords.length, GeometryArray.COORDINATES, indices.length);
geometry.setCoordinates(0, coords);
geometry.setCoordinateIndices(0, indices);

Shape3D shape = new Shape3D(geometry);

Transform3D rotacion = new Transform3D();

 rotacion.rotY(0.7);

TransformGroup trans = new TransformGroup(rotacion);

trans.addChild(shape);
root.addChild(trans);

return root;
...
```

Geometria avançada

Donat que amb les classes exposades no es poden representar tots els objectes, augmentem el nivell de complexitat i presentem un conjunt de classes, que combinades entre si es pot representar objectes complexes i de formes complicades i a més a més preparar-los per a una millor *renderització*.

- *GeometryInfo*: Classe que s'utilitza de suport per a dades (vèrtexs) i poder-hi aplicar diferents eines, p.e. indexar els vèrtexs, per a una millor representació i *renderització*. Posteriorment es pot extreure la geometria de l'objecte per a utilitzar-la en un objecte *Shape3D*. L'únic paràmetre no conegut que admet el seu constructor és un constant (*int*) que accepta les següents opcions:
    - POLYGON_ARRAY: Declara que l'objecte que en sortirà no tindrà un forma estàndard.
    - QUAD_ARRAY: Especifica que tindrà forma de quadrat.
    - TRIANGLE_ARRAY: Declara que l'objecte tindrà forma de triangle.
    - TRIANGLE_FAN_ARRAY: Especifica que tindrà forma de triangle units a un vèrtex central.
    - TRIANGLE_STRIP_ARRAY: Especifica que tindrà forma de triangle units un al costat de l'altre, com en una malla.
- *Triangulator*: Aquesta classe s'usa per dividir un polígon complex en triangles. Aquesta classe només funciona aplicat a un objecte

GeometryInfo amb l'opció POLYGON_ARRAY. En la versió de Java3D (v.1.4.0), no fa falta utilitzar-la, ja s'aplica automàticament sobre l'objecte *GeometryInfo* en utilitzar l'opció POLYGON_ARRAY.

- *NormalGenerator*: Classe que s'empra per a la generació dels vectors normals per a cada vèrtex que definiran la suavitat o rugositat de l'objecte. Aquests vectors s'utilitzen per determinar la il·luminació dels objectes d'una forma acurada. Un paràmetre important és el que determina l'angle (per defecte 44º) a partir del qual és considera que hi ha un plec.
- *Stripifier*: Aquesta classe s'utilitza per a formar tires de triangles donada una cadena inconnexa de triangles. L'objectiu és l'optimització del temps de càlcul del processador a l'hora del *renderitzat*.

La forma i l'ordre d'aplicació de les anteriors classes és el següent.

1. Definir l'objecte *GeometryInfo*, mitjançant els mètodes que incorpora.
2. Aplicar sobre l'anterior objecte les classes anteriors si és necessari. Amb els mètodes:
    a. *void triangulate(GeometryInfo* gi*)*. Per a *Triangulator*.
    b. *void generateNormals(GeometryInfo* gi*)*.
    Per a *NormalGenerator*.
    c. *void stripify(GeometryInfo* gi*)*. Per a *Stripifier*.
3. Després de l'aplicació de les eines anteriors, l'objecte *GeometryInfo* està preparat per extreure-li la geometria de la figura desitjada. Això s'aconsegueix mitjançant els mètodes o semblants:
    a. *GeometryArray getGeometryArray()*. Per extreure la geometria de l'objecte.
    b. *IndexedGeometryArray getIndexedGeometryArray()*. Per extreure la geometria indexada de l'objecte.
4. La geometria obtinguda s'utilitza en la creació (o modificació) d'un objecte *Shape3D* i una vegada definida l'aparença s'aconsegueix l'objecte desitjat.

Un exemple de *GeometryInfo*:

```
BranchGroup createSceneGraph() {

        BranchGroup objRoot = new BranchGroup();

        Transform3D rotate = new Transform3D();
        rotate.rotY(Math.PI/6.0d);
        TransformGroup objRotate = new TransformGroup(rotate);


        TriangleStripArray tfa;
        int     N = 6;          // número de caras
        int     totalN =2*N;
        int     stripCounts[] = {totalN};
        int     n=0;
        Point3f coords[] = new Point3f[totalN];
        float   r = 0.2f;
        float   x, y,z;

        for(n= 0; n < totalN;  n=n+2 ){
                    x = (float)(r*Math.sin(Math.PI/(N-1)*n));
                    z = (float)(r*Math.cos(Math.PI/(N-1)*n));

                    coords[n] = new Point3f(x, -0.5f, z);
                    coords[n+1] = new Point3f(2*x, 0.5f, 2*z);
        }

        GeometryInfo gi =new GeometryInfo (GeometryInfo.TRIANGLE_STRIP_ARRAY);
        gi.setCoordinates(coords);
        gi.setStripCounts(stripCounts);

        NormalGenerator ng = new NormalGenerator();
        ng.setCreaseAngle((float) Math.toRadians(10)); // ángulo de pliegue
        ng.generateNormals(gi);

        Stripifier st = new Stripifier();
        st.stripify(gi);

        GeometryArray figura = gi.getGeometryArray();

        Shape3D shape = new Shape3D(figura,app());

        …
```

Loaders

Si finalment amb les classes exposades no s'aconsegueix representar l'objecte que es vol, Java3d proporciona una base per a utilitzar *Loaders* (carregadors).

La biblioteca proporciona dues interfícies (*Loader* i *Scene*) i dues classes (*LoaderBase* i *SceneBase*) com a base per a poder utilitzar *Loaders*. Els *Loaders* ens permeten importar figures en 3D dibuixades en altres aplicacions externes (3DStudio, AutoCAD, etc.). Per aconseguir això és necessari definir una classe *Loader* específica (subclasse) per al format que es vol importar. *Loader* s'encarrega de localitzar i capturar les característiques del fitxer que conté la imatge i ha de saber com llegir-les. *Scene* emmagatzemarà tota la informació capturada, per posteriorment ésser extreta per a poder-la representar. Per ampliar la informació sobre els *loader* vegeu [10].

Java3D incorpora dos carregadors com a exemples: *Lw3dLoader* (per a fitxers *Lightwave* 3D) i *ObjectFile* (per a fitxers *Wavefront*).

Un exemple de com s'utilitza un *loader* (*ObjectFile*):

```
import java.applet.Applet;
import java.awt.*;
import java.util.*;
import java.io.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.loaders.*;
import com.sun.j3d.loaders.objectfile.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.SimpleUniverse;

public class Cargador extends Applet {

   private java.net.URL filename;

   public Cargador(java.net.URL url) {
   filename = url;

   GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
   Canvas3D canvas = new Canvas3D(config);
   this.setLayout(new BorderLayout());
   this.add(canvas, BorderLayout.CENTER);
   SimpleUniverse universe = new SimpleUniverse(canvas);
   universe.getViewingPlatform().setNominalViewingTransform();
   BranchGroup scene =createPlane();
   universe.addBranchGraph(scene);

   }

   public void init() {
   if (filename == null) {

      try {
               java.net.URL path = getCodeBase();
               filename = new java.net.URL(path.toString() + "./MustangAirplane.obj");
      }
```

```
        catch (java.net.MalformedURLException ex) {
                System.err.println(ex.getMessage());
                ex.printStackTrace();
                System.exit(1);
        }
    }
    }

private Appearance app() {

    Appearance app=new Appearance();
    PolygonAttributes polyAtt=new PolygonAttributes();
    polyAtt.setCullFace(PolygonAttributes.CULL_NONE);
    polyAtt.setPolygonMode(PolygonAttributes.POLYGON_LINE);
    app.setPolygonAttributes(polyAtt);

    Material mat=new Material();
    app.setMaterial(mat);

    return app;
    }
BranchGroup createPlane() {
    java.net.URL PlaneURL = filename;


    int flags = ObjectFile.RESIZE;
        ObjectFile f = new ObjectFile(flags);
        Scene s = null;
        try {
            s = f.load(PlaneURL);
        }
        catch (Exception e) {
            System.err.println(e);
            System.exit(1);
        }

    Group sceneGroup = s.getSceneGroup();

    Hashtable namedObjects = s.getNamedObjects();
    Enumeration e =  namedObjects.keys();
    while (e.hasMoreElements()) {
        String name = (String) e.nextElement();

        Shape3D shape = (Shape3D) namedObjects.get(name);

        shape.setAppearance(app());
    }

    BranchGroup retVal = new BranchGroup();
    BoundingSphere bounds = new BoundingSphere(new Point3d(), 100.0);

    PointLight poLi=new PointLight();
    poLi.setPosition(new Point3f(0.0f,0.0f,0.0f));
    poLi.setInfluencingBounds(bounds);
    poLi.setColor(new Color3f(1f,1f,0f));
    retVal.addChild(poLi);

    DirectionalLight light =
    new DirectionalLight( new Color3f(1f,1f,0.5f),
    new Vector3f(0.0f, -1.0f, 0.0f) );
    light.setInfluencingBounds(bounds);
    retVal.addChild(light);

    DirectionalLight light2 =
    new DirectionalLight( new Color3f(0.5f,0.5f,0.2f),
    new Vector3f(0.0f, 1.0f, -0.5f) );
    light2.setInfluencingBounds(bounds);
    retVal.addChild(light2);

    AmbientLight alight= new AmbientLight();
    retVal.addChild(alight);

    retVal.addChild(s.getSceneGroup());
    return retVal;
    }
```

Aparences

Una vegada presentades les diverses formes de definir objectes geomètrics, és convenient veure com veurà l'usuari els objectes que vol representar.

Com ja s'ha dit, la classe *Appearance* són nodes del tipus *NodeComponent*; això significa que cada nou objecte aparença pot estar associat a molts objectes geomètrics diferents.

Les característiques modificables d'un objecte *Appearance* s'anomenen **atributs** i són subclasses del tipus *NodeComponent*. Per a llegir o modificar cada un dels varis atributs en temps d'execució cal tenir activades les **capacitats** corresponents.

P.e.: aparença.*setCapability(*ALLOW_COLORING_ATTRIBUTES_READ*)*

Els atributs són:

- *ColoringAttributes*: S'utilitza per definir els colors que s'utilitzaran en l'objecte i la seva ombra.
- *LineAttributes*: Defineix l'estil de línia i el seu grossor.
- *PointAttributes*: Defineix les característiques del punt.
- *PolygonAttributes*: Defineix com es representarà un polígon.
- *RenderingAttributes*: Defineix les característiques de *renderització* comuns a tots els tipus (classes) primitius.
- *TransparencyAttributes*: Determina el tipus i el grau de transparència dels objectes associats a l'aparença.
- *Material*: Determina el comportament de la superfície dels objectes associats a l'aparença al ser il·luminat.
- *TextureAttributes*: Configura el tipus de textura i la seva col·locació sobre l'objecte.
- *Texture*: Defineix la imatge a utilitzar com a textura i els filtres que poden aplicar-se sobre ella.
- *TexCoordGeneration*: Configura els paràmetres que controlen la generació automàtica de textures.
- *TextureUnitState*: Configura els paràmetres que controlen la col·locació d'una unitat de textura. I la possible col·locació de vàries d'elles.

La forma d'aplicació de cada un dels atributs és a base de mètodes **set/get** específics.

Dos exemples de mètode de definició d'aparença:

```
private Appearance app() {

        Appearance app=new Appearance();

        PolygonAttributes polyAtt=new PolygonAttributes();
        polyAtt.setCullFace(PolygonAttributes.CULL_FRONT);
        polyAtt.setPolygonMode(PolygonAttributes.POLYGON_LINE);
        polyAtt.setPolygonOffset(1.0f);
        polyAtt.setBackFaceNormalFlip(true);
        app.setPolygonAttributes(polyAtt);

        ColoringAttributes colorAtt=new ColoringAttributes(0.0f,1.0f,0.0f,ColoringAttributes.SHADE_GOURAUD);
        colorAtt.setShadeModel(ColoringAttributes.SHADE_GOURAUD);
        app.setColoringAttributes(colorAtt);

        PointAttributes pointAtt=new PointAttributes();
        pointAtt.setPointSize(6.0f);
        pointAtt.setPointAntialiasingEnable(true);
        app.setPointAttributes(pointAtt);

        LineAttributes lineAtt=new LineAttributes();
        lineAtt.setLinePattern(LineAttributes.PATTERN_DASH);
        app.setLineAttributes(lineAtt);

        return app;
}

private Appearance app() {

        Appearance app=new Appearance();

        PolygonAttributes polyAtt=new PolygonAttributes();
        polyAtt.setCullFace(PolygonAttributes.CULL_NONE);
        polyAtt.setPolygonMode(PolygonAttributes.POLYGON_FILL);
        polyAtt.setPolygonOffset(1.0f);
        polyAtt.setBackFaceNormalFlip(true);
        app.setPolygonAttributes(polyAtt);

        TransparencyAttributes transAtt=new TransparencyAttributes();
        transAtt.setTransparencyMode(TransparencyAttributes.BLENDED);
        transAtt.setTransparency(0.2f);
        app.setTransparencyAttributes(transAtt);

        Material mat=new Material();
        app.setMaterial(mat);

        return app;
}
```

Il·luminació

Per a poder representar el més semblant possible un objecte real en un univers virtual s'han d'usar les il·luminacions.

Per a poder il·luminar un objecte i que es vegi un resultat calen dues coses. La primera és la inserció d'un llum en el diagrama d'escena i la segona, essencial, cal definir l'atribut *Material* de l'aparença associada a l'objecte il·luminat i la geometria dels objectes ha de tenir calculats els vectors Normals per a cada vèrtex.

Els tipus de focus de llum que pot definir Java3D són els següents:

- Ambient: El raig de llum d'aquesta classe està repartit per tot l'espai i la intensitat és igual en tot punt de la zona d'influència. Classe *AmbientLight*.
- Direccional: Tots els rajos de llum van en una mateixa direcció, simula els rajos del Sol. Classe *DirectionalLight*.
- Punt de llum: Els rajos surten d'un punt concret de l'espai en totes direccions com en el cas de la bombeta. Classe *PointLight*.
- Focus de llum: Un cas particular de l'anterior és la llum que surt d'una llanterna. Classe *Spotlight* subclasse de *Pointlight*.

Aquestes classes són subclasses d'una classe abstracta anomenada *Light*, que defineix una sèrie de propietats (color de la llum, estat, zona d'influència) així com una col·lecció de referències a tots els Nodes que vol il·luminar. Per a poder accedir tots aquests paràmetres en temps d'execució cal activar les capacitats específiques (p.e. ALLOW_COLOR_WRITE). Cal també destacar d'aquesta classe que incorpora uns mètodes anomenats "*Scope*" (*addScope, insertScope*, etc.) els quals s'utilitzen per a restringir els Nodes als quals afectarà la il·luminació i per tenir un control com si fos una llista d'objectes.

En l'exemple de *Loaders* hi ha un exemple dels tipus d'il·luminació.

Material

La classe *Material*, atribut de *Appearance*, engloba els diferents tipus de reflexió a les possibilitats de llum: ambiental (*ambient*), emissiva (*emissive*), difusa (*diffuse*) i especular (*specular*) i també la brillantor (*shininess*) del propi objecte representat. Per tant el nou objecte *Material* ha de definir el color que prendrà el reflex per a cada un de les possibles llums.

Ambient: Aquest tipus de reflexió es produeix quan l'objecte està il·luminat per una llum ambiental. El reflex és uniforme en tota la superfície de l'objecte.

Emissive: L'objecte reflecteix el color propi de la seva superfície (o el que se li defineixi) i no fa falta que estigui il·luminat. S'utilitza per simular el Sol, bombetes, etc.

Diffuse: Aquesta reflexió produeix un degradat del color blanc (on incideix el focus de llum) cap al color de l'objecte. Es produeix quan la llum incideix en una zona ampla de l'objecte (part blanca). Si l'objecte té ombres el degradat arriba fins al negre de l'ombra.

Specular: Aquesta reflexió es produeix en el lloc que incideix un focus de llum concentrat en un punt.

Shininess: Aquest paràmetre defineix la mida de la zona de brillantor d'una llum que incideix en un punt concret.

Un exemple d'ús de la classe *Material*:

```
import java.applet.*;
import java.awt.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.geometry.Sphere;

public class Combinacion_Material extends Applet {
  public Combinacion_Material() {
    GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas = new Canvas3D(config);
    this.setLayout(new BorderLayout());
    this.add(canvas, BorderLayout.CENTER);

    SimpleUniverse universe = new SimpleUniverse(canvas);
    universe.getViewingPlatform().setNominalViewingTransform();

    universe.addBranchGraph(createSceneGraph());
  }

  private BranchGroup createSceneGraph() {
    BranchGroup root = new BranchGroup();

    DirectionalLight light =
      new DirectionalLight( new Color3f(1.0f, 1.0f, 1.0f),
                    new Vector3f(0.5f, 0.5f, -0.5f) );
    BoundingSphere bounds = new BoundingSphere(new Point3d(), 100.0);
    light.setInfluencingBounds(bounds);
    root.addChild(light);

    Material material = new Material();
    material.setDiffuseColor( new Color3f(0.37f, 0.37f, 0.37f) );
    material.setSpecularColor( new Color3f(0.89f, 0.89f, 0.89f) );
    material.setShininess(17.0f);
```

```
    material.setAmbientColor( new Color3f(0.8f, 0.8f, 0.8f) );
    Appearance app = new Appearance();
    app.setMaterial(material);

    Sphere sphere = new Sphere(0.6f, app);
    root.addChild(sphere);

    return root;
  }

  public static void main(String[] args) {
    Combinacion_Material applet = new Combinacion_Material();
    Frame frame = new MainFrame(applet, 256, 224);
  }
}
```

Ombres

Si Java3D permet definir il·luminacions i veure l'efecte sobre els objectes, és gairebé necessari que també permeti definir l'ombra que projectaran aquests objectes il·luminats. La creació d'ombres amb Java3D no és gens fàcil perquè la biblioteca no ofereix una forma directa de fer-ho, però algunes opcions són:

- Una opció seria, si l'objecte té d'altres objectes a prop, mitjançant els mètodes "*scope*" amagar les parts on és projectaria l'objecte del qual volem simular l'ombra.

- L'alternativa més utilitzada és la de crear un objecte de color negre sense que la seva aparença tingui atribut *Material*, la geometria (o textura) del qual seria la projecció de l'objecte del que representa l'ombra sobre un pla.

  Un exemple de classe per simular l'ombra i que crea la projecció d'un objecte passat com a paràmetre i que necessita a més a més el vector director de la il·luminació, el color de l'ombra (normalment fosc) i l'alçada de l'objecte fins l'ombra.

```
// Define a Shadow Polygon Class
public class SimpleShadow extends Shape3D {

    SimpleShadow(GeometryArray geom, Vector3f direction, Color3f col, float height) {

        int vCount = geom.getVertexCount();
        QuadArray poly = new QuadArray(vCount, GeometryArray.COORDINATES
                | GeometryArray.COLOR_3);
        int v;
        Point3f vertex = new Point3f();
        Point3f shadow = new Point3f();
        for (v = 0; v < vCount; v++) {
```

```
                geom.getCoordinate(v, vertex);
                shadow.set(vertex.x + (vertex.y - height) * direction.x, height + 0.0001f, vertex.z +
                    (vertex.y - height) * direction.y);
                poly.setCoordinate(v, shadow);
                poly.setColor(v, col);
            }
            this.setGeometry(poly);

        }

    }
```

Textures

En el cas que es vulgui representar que un objecte sigui d'un cert tipus de material, Java3D dona suport a utilitzar textures per simular la superfície dels objectes físics.

Els passos principals per incorporar una textura són:

- Carregar la mostra de la textura.
- Establir els atributs.
- Afegir la textura a l'aparença.
- Calcular les coordenades de la textura (*téxels*) per correspondre'ls amb les de l'objecte.
- Assignar l'aparença a l'objecte.

Carregar les textures

Aquest primer pas consisteix a carregar la textura en una subclasse de la classe abstracta *Texture*. Poden ser *Texture2D* (per a objectes 2D), *Texture3D* (per a objectes 3D) i *TextureCubeMap* (per a objectes cúbics). Per a fer-ho utilitzarem la classe *TextureLoader* per carregar el fitxer imatge; i posteriorment, mitjançant els mètodes que incorpora, extreure un objecte *ImageComponent2D* (acotat, escalat o escala real) per incorporar-lo després a l'objecte *Texture* (amb el mètode *Texture.setImage*).

Filtres de textura

Els filtres de textura són una col·lecció de propietats que incorpora la classe *Texture*. Serveixen per a definir la manera de com s'estendrà la textura per sobre la figura, la resolució i d'altres dades útils. Alguns dels més importants són:

1. Aquest filtre tracta de definir l'expansió de la textura en els eixos de coordenades de les textures (t i s), anomenades *téxels*, i només poden agafar valors entre 0 i 1. Per definir aquest comportament, la classe *Texture* proporciona els mètodes següents:

   - *void setBounderyModeS*(*int* bounderyModeS)
   - *void setBounderyModeT*(*int* bounderyModeT)

   Les possibles opcions per a tots dos mètodes són:
   - WRAP: Es repetirà la textura en aquelles parts de l'objecte fora del rang (0,1).
   - CLAMP: S'estendran els colors de la textura (última fila i columna) quan sigui fora del rang (0,1).
   - CLAMP_TO_BOUNDARY i CLAMP_TO_EDGE: De manera similar a CLAMP, però el primer s'estendrà fins omplir la resta de la cara i el segon fins a tocar una aresta.
2. En segon lloc hi ha un filtre que s'utilitza per controlar la resolució o qualitat de la textura a l'hora de reduir o ampliar l'escala de la textura. Per definir aquest comportament s'utilitzen els mètodes següents de la classe *Texture*.
   - *void setMagFilter*(*int* magFilter)
   - *void setMinFilter*(*int* minFilter)

   En tots dos casos s'utilitzen els mateixos valors, alguns dels quals són:

   - BASE_LEVEL_POINT: El *píxel* de la figura geomètrica pren el color del *téxel* més proper. Aquesta opció dona un resultat bastant pobre, sense detalls i amb salts de colors. És equivalent a FASTEST.
   - BASE_LEVEL_LINEAR: El *píxel* de la figura pren el color de la mitjana ponderada dels colors dels quatre *téxels* més propers. És

equivalent a NICEST. El resultat és un degradat més suau. És l'opció més utilitzada.

Cal destacar que en el cas de la reducció d'escala, com que la malla de *téxels* és més densa que la malla de *píxels*, les opcions BASE_LEVEL_LINEAR o NICEST no generen un resultat prou òptim. Java3D soluciona aquest contratemps utilitzant una tècnica anomenada *MIPmap*, que consisteix en emmagatzemar, a l'hora de carregar la textura, una col·lecció de textures de diferents escales en potències de 2. Per a utilitzar aquesta tècnica cal haver carregat la textura (*TextureLoader*) amb l'opció GENERATE_MIPMAP i utilitzar una de les dues alternatives equivalents a les opcions anteriors:

- MULTI_LEVEL_POINT
- MULTI_LEVEL_LINEAR

Alguns exemples d'inclusió d'una textura:

```
Appearance app(){

    Appearance app = new Appearance();

    PolygonAttributes polyAttrib = new PolygonAttributes();
    polyAttrib.setCullFace(PolygonAttributes.CULL_NONE);
    app.setPolygonAttributes(polyAttrib);

    TexCoordGeneration texCoord = new TexCoordGeneration(
                                        TexCoordGeneration.OBJECT_LINEAR,
                                        TexCoordGeneration.TEXTURE_COORDINATE_2);
    app.setTexCoordGeneration(texCoord);

    String filename = "seco.jpg";

    TextureLoader loader = new TextureLoader(filename, this);
    ImageComponent2D image = loader.getImage();

    Texture2D texture = new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
                    image.getWidth(), image.getHeight());
    texture.setImage(0, image);
    texture.setEnable(true);
    texture.setBoundaryModeS(Texture.WRAP);
    texture.setBoundaryModeT(Texture.CLAMP);

    app.setTexture(texture);

    return app;
}



private Appearance app() {

        Appearance app=new Appearance();

        PolygonAttributes polyAtt=new PolygonAttributes();
```

```
polyAtt.setCullFace(PolygonAttributes.CULL_NONE);
polyAtt.setPolygonMode(PolygonAttributes.POLYGON_FILL);
polyAtt.setPolygonOffset(1.0f);
polyAtt.setBackFaceNormalFlip(true);
app.setPolygonAttributes(polyAtt);

Material mat=new Material();
app.setMaterial(mat);

// Texture2D

Texture2D          texture;
String imageFile;
String codeBaseString=null;
String texImage;

texImage = "Fract.gif";

TextureLoader texLoader = new TextureLoader(texImage,new String("RGBA"),
        TextureLoader.GENERATE_MIPMAP, this);

texture = (Texture2D) texLoader.getTexture();
texture.setBoundaryColor(new Color4f());
texture.setBoundaryModeS(Texture.WRAP);
texture.setBoundaryModeT(Texture.WRAP);
texture.setEnable(true);
texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);

app.setTexture(texture);

 TexCoordGeneration texGen = new
                TexCoordGeneration(TexCoordGeneration.OBJECT_LINEAR,
                TexCoordGeneration.TEXTURE_COORDINATE_2, new Vector4f(1.0f,
                0.0f, 0.0f, 0.0f), new Vector4f(0.0f, 1.0f, 0.0f, 0.0f));
texGen.setCapability(TexCoordGeneration.ALLOW_ENABLE_WRITE);
texGen.setEnable(true);
app.setTexCoordGeneration(texGen);


return app;
}
```

Atributs de Textura

La classe *TextureAttributes* s'encarrega de controlar la manera com la textura serà aplicada sobre una superfície. Aquesta classe és del tipus *NodeComponent*, per tant pot ser compartida per varis objectes *Appearance*. Alguns dels atributs són: el mode de textura, l'objecte *Transform* de com s'aplica la textura sobre l'objecte, atributs que guarden colors per a diferents objectius, un corrector de perspectiva i una taula on guarda tots els colors de la textura, etc.

TextureMode

Defineix com es mesclaran els colors de la textura i de l'objecte. S'aplica mitjançant el mètode:

- *void setTextureMode*(*int* textureMode)

Pot prendre diferents valors:

- *REPLACE*: Valor per defecte. El color de la textura sobreposa qualssevol altre color (el de l'objecte) i efecte modificador com ara possibles il·luminacions.
- *MODULATE*: Mescla tots els colors i efectes sobre l'objecte i la textura.
- *DECAL*: A part del color de la textura només té en compte la possible transparència de la textura. Si la textura no té transparència és equivalent a *REPLACE*.
- *BLEND*: Actua com *MODULATE* però a més a més incorpora a la mescla un color (atribut *Blend color*). És l'opció més òptima però la més costosa de processar.
- *COMBINE*: opció avançada per a crear una mescla totalment personalitzada. Justament amb d'altres atributs (*Combine mode*, *Combine scale factor*, etc.) es pot aconseguir una mescla determinada.

Transform

Aquest atribut emmagatzema l'objecte *Trasform3D* que determina la posició de la textura en l'univers. Sobre aquest objecte es poden aplicar qualssevol transformació que s'ha explicat. S'aplica mitjançant el mètode:

- *void setTextureTransform*(*Trasform3D* transform)

Correcció de perspectiva

Aquest atribut s'encarrega de corregir possibles errors en la perspectiva de la textura quan l'objecte on se li ha aplicat la textura canvia de posició o perspectiva respecte l'observador. S'aplica mitjançant el mètode:

- *void setPerspetiveCorrectionMode*(*int* mode)

Les opcions són NICEST i FASTEST, segons es vulgui obtenir millor qualitat o rapidesa respectivament. Per defecte és NICEST ja que no comporta gaire càrrega computacional.

Taula de colors

Aquest atribut guarda una taula amb tots els colors de la textura abans d'aplicar-la. Pot ser modificada.

Un exemple d'atributs de textura:

```
private Appearance app() {

   Appearance app=new Appearance();

   PolygonAttributes polyAtt=new PolygonAttributes();
   polyAtt.setCullFace(PolygonAttributes.CULL_NONE);
   polyAtt.setPolygonMode(PolygonAttributes.POLYGON_FILL);
   polyAtt.setPolygonOffset(1.0f);
   polyAtt.setBackFaceNormalFlip(true);
   app.setPolygonAttributes(polyAtt);

   Material mat=new Material();
   //mat.setEmissiveColor(new Color3f (0.0f, 0.0f, 1.0f));
   app.setMaterial(mat);


   // Texture2D

   Texture2D  texture;
   String  texImage = "lineaAma.jpg";


   TextureLoader texLoader = new TextureLoader(texImage,
            new String("RGBA"),TextureLoader.GENERATE_MIPMAP, this);


   texture = (Texture2D) texLoader.getTexture();

   texture.setBoundaryColor(new Color4f());
   texture.setBoundaryModeS(Texture.WRAP);
   texture.setBoundaryModeT(Texture.WRAP);
   texture.setEnable(true);
   texture.setMinFilter(Texture.BASE_LEVEL_LINEAR);
   texture.setMagFilter(Texture.BASE_LEVEL_LINEAR);

   app.setTexture(texture);


   // TextureAttributes

   TextureAttributes textureAttr = new TextureAttributes();
   textureAttr.setTextureMode(TextureAttributes.REPLACE);
   Transform3D rotate = new Transform3D();
   rotate.rotZ(Math.PI);
   textureAttr.setTextureTransform(rotate);
   app.setTextureAttributes(textureAttr);

   TexCoordGeneration texGen = new TexCoordGeneration(TexCoordGeneration.OBJECT_LINEAR,
                        TexCoordGeneration.TEXTURE_COORDINATE_2,
                        new Vector4f(1.0f, 0.0f, 0.0f, 0.0f), new Vector4f(0.0f, 1.0f, 0.0f, 0.0f));
```

```
texGen.setCapability(TexCoordGeneration.ALLOW_ENABLE_WRITE);
texGen.setEnable(true);
app.setTexCoordGeneration(texGen);

return app;
}
```

Generació automàtica de coordenades de textura

Una vegada carregada i parametritzada la textura cal fer coincidir cada vèrtex de la textura, *tèxel*, amb un vèrtex de l'objecte sobre el qual la volem col·locar. Fer-ho manualment és possible amb els mètodes *setTextureCoordinate* de les classes per a definir geometries (*GeometryArray*, *IndexedGeometryArray* i *GeometryInfo*). Però com és molt laboriós Java3D proporciona la classe *TexCoordGeneration* i *Appearance* conté el mètode *setTexCoordGeneration* per automatitzar el procés.

Els paràmetres més importants que necessita la classe *TexCoordGeneration* són el mode de generació i el format. Els exemples anteriors de textures exemplifiquen la generació automàtica de coordenades de textura.

El mode de generació

Alguns dels valors que accepta són OBJECT_LINEAR, EYE_LINEAR i SPHERE_MAP. El sufix LINEAR significa que calcula els téxels mitjançant projeccions lineals sobre uns plans que crea automàticament.

- OBJECT_LINEAR: L'efecte visual és que quan l'objecte es mou (p.e. gira) sembla que la textura també es mogui.
- EYE_LINEAR: En aquesta opció la textura només sembla que es mogui quan canvia el punt de vista (les coordenades) de l'observador.
- SPHERE_MAP: Simula l'efecte reflector que es dóna en els objectes esfèrics. Utilitza els vectors Normals (cal haver-los activat en la geometria) i les coordenades del punt de vista de l'observador.

Format

L'atribut de *format* determina la dimensió de la textura que es vol generar. Pot prendre els valors TEXTURE_COORDINATE_2, canviant el 2 pel 3 ó 4 segons sigui necessari.

Boira

En un terreny més ambiental, trobem l'efecte "boira".

El concepte de "boira", definit per la classe abstracta *Fog*, engloba dos efectes:

- Primerament ens trobem amb la classe *LinearFog*. L'ús d'aquesta classe provoca que els colors dels objectes geomètrics de l'espai es barregin amb el color de les partícules de la "atmosfera". Això provoca una certa atenuació dels colors. La classe permet definir el color de les partícules i la zona d'influència de la "boira".
- El segon efecte és el de la "boira" pròpiament. La classe que la simula és *ExponentialFog*. Aquesta classe permet definir el color de la boira, la seva densitat i la zona d'influència.

Ambdós efectes han de ser utilitzats com a fills del Node arrel per a que afectin a tot l'univers representat. I per a un millor resultat el color de l'efecte ha de ser el mateix que el color de fons.

Dos exemples de *LinearFog* i *ExponentialFog*, respectivament:

```
...
void  createSceneGraph() {
   BoundingSphere bounds = new BoundingSphere(new Point3d(), Double.MAX_VALUE);

   DirectionalLight azulClaro =
     new DirectionalLight( new Color3f(0.5f, 0.7f, 1.0f), new Vector3f(0.0f, 0.0f, -1.0f) );
   azulClaro.setInfluencingBounds(bounds);
   objRoot.addChild(azulClaro);

   Background bg = new Background(new Color3f(0.9f,0.9f,1.0f));
   bg.setApplicationBounds(bounds);
   objRoot.addChild(bg);

   LinearFog nieblaLineal=new LinearFog(new Color3f(0.9f,0.9f,1.0f),2.0f,20.0f);
   nieblaLineal.setInfluencingBounds(bounds);
   objRoot.addChild(nieblaLineal);
```

```
}...

...
void  createSceneGraph() {
   BoundingSphere bounds = new BoundingSphere(new Point3d(), Double.MAX_VALUE);

   DirectionalLight azulClaro =
     new DirectionalLight( new Color3f(0.5f, 0.7f, 1.0f), new Vector3f(0.0f, 0.0f, -1.0f) );
   azulClaro.setInfluencingBounds(bounds);
   objRoot.addChild(azulClaro);

   Background bg = new Background(new Color3f(0.9f,0.9f,1.0f));
   bg.setApplicationBounds(bounds);
   objRoot.addChild(bg);

   ExponentialFog nieblaExp=new ExponentialFog(new Color3f(0.9f,0.9f,1.0f),3f);
   nieblaExp.setInfluencingBounds(bounds);
   objRoot.addChild(nieblaExp);


}...
```

Efectes Sonors

Continuant amb efectes ambientals una forma de millorar una escena i de fer-la més real és incloent-hi sons. Java3D posa la base per a crear efectes sonors en la superclasse abstracta *Sound*. Les seves subclasses es diferencien en l'origen del so.

Per als sorolls de fons, sense punt d'origen, s'utilitza la classe *BackgroundSound*. I per a sorolls amb punt d'origen *PointSound* i la seva subclasse *ConeSound* (sons direccionals en forma de con).

El procediment habitual per incorporar efectes sonors és:

- Crear una referència a l'equip d'àudio (classe *AudioDevice*) de l'equip informàtic a través de l'objecte *Viewer*, dins de la classe *SimpleUnivers*, i el seu mètode *createAudioDevice*. Mitjançant aquest lligam Java3D podrà calcular les modificacions a efectuar sobre l'efecte sonor ja que sabrà on està col·locat l'observador (objecte *ViewingPlatform*, també accessible des de *SimpleUnivers*).
- Definir la zona d'influència (subclasses de *Bounds*) de l'efecte sonor.
- Crear l'objecte *MediaContainer* per a carregar el fitxer del so.
- Crear el so mitjançant alguna de les classes proposades i activar-lo amb el mètode *setEnable*.

- Configurar les repeticions o fer-lo infinit mitjançant el valor INFINITE_LOOP.
- Finalment cal penjar-lo de la branca arrel i tractar-ho com un Node.

En la creació del *MediaContainer* i de l'efecte sonor cal tenir en compte que cal activar totes les capacitats d'ambdues classes ja que Java3D modifica els seus atributs en temps d'execució per tal d'adaptar l'efecte a la posició de l'observador/oient.

Un exemple d'ús d'efectes sonors:

```
...
try {
        url = new URL("file:./tecno.au");
    }
        catch (Exception e) {
        System.out.println(e.getMessage());

    }
MediaContainer sonidoMedia=new MediaContainer();
sonidoMedia.setCapability(MediaContainer.ALLOW_URL_WRITE);
sonidoMedia.setCapability(MediaContainer.ALLOW_URL_READ);
sonidoMedia.setURLObject(url);
sonidoMedia.setCacheEnable(false);

BackgroundSound bgSound=new BackgroundSound();
bgSound.setCapability(Sound.ALLOW_SOUND_DATA_READ);
bgSound.setCapability(Sound.ALLOW_ENABLE_WRITE);
bgSound.setCapability(Sound.ALLOW_INITIAL_GAIN_WRITE);
bgSound.setCapability(Sound.ALLOW_SOUND_DATA_WRITE);
bgSound.setCapability(Sound.ALLOW_SCHEDULING_BOUNDS_WRITE);
bgSound.setCapability(Sound.ALLOW_CONT_PLAY_WRITE);
bgSound.setCapability(Sound.ALLOW_RELEASE_WRITE);
bgSound.setCapability(Sound.ALLOW_DURATION_READ);
bgSound.setCapability(Sound.ALLOW_IS_PLAYING_READ);
bgSound.setCapability(Sound.ALLOW_LOOP_WRITE);

bgSound.setSoundData(sonidoMedia);
bgSound.setSchedulingBounds(bounds);
bgSound.setEnable(true);
bgSound.setLoop(Sound.INFINITE_LOOPS);
objRoot.addChild(bgSound);

return objRoot;
...
```

Relacions amb l'usuari

Aquest apartat tracta de la manera com respondrà o com volem que reaccioni l'aplicació a una acció determinada de l'usuari.

Java3D proporciona un conjunt de classes, subclasses de la classe abstracta *Behavior* que s'encarreguen de controlar la resposta segons l'estímul.

Algunes de les més importants són:

- *Billboard*: s'encarrega de controlar el canvi d'orientació dels objectes degut al moviment del punt de vista de l'observador.
- LOD: Sigles d'una classe que signifiquen "nivell de detall" (*Level Of Detail*), la seva subclasse *DistanceLOD* s'encarrega de regular la qualitat de detall dels objectes representats en escena. Utilitza grups *Switch*, on cada node representa una qualitat a diferent distancia respecte l'observador.
- *Interpolator*: Controla la resposta de l'aplicació al pas del temps. Aquesta classe i les seves subclasses estan dissenyades per a crear animacions.
- *MouseBehavior*: Les seves subclasses controlen les respostes generades per accions del ratolí. Implementa diverses interfícies de Java2.
- *KeyNavigatorBehavior*: S'utilitza per a definir, juntament amb *KeyNavigator*, les respostes a la utilització de les tecles de direcció del teclat i canviar la posició de l'observador.

Crear una classe *Behavior*

Si les subclasses de *Behavior* no són suficients per definir un cas particular; crear la nostra pròpia classe *Behavior* (subclasse de *Behavior*) és una opció molt utilitzada.

Tota classe d'aquest tipus ha d'implementar un constructor, un mètode d'inicialització (*initialize*) i un mètode de resposta (*processStimulus*). També ha de tenir referenciat l'objecte al qual s'aplica la resposta. Aquest objecte es referencia en el constructor de la nova classe *Behavior*.

El mètode d'inicialització és on queda definida la condició d'activació, i es crida automàticament en el moment de la representació de la Rama de Contingut.

El mètode de resposta (*processStimulus*) és on es defineix la resposta segons l'acció i si cal es redefineix la condició d'activació.

La condició d'activació es defineix utilitzant el mètode *wakeupOn*(*WakeupCondition*).

Per a poder utilitzar aquesta o alguna de les classes *Behavior* cal tenir present:

- S'ha de definir la zona d'influència (*SchedulingBounds*).
- S'ha d'haver activat les capacitats de READ i WRITE sobre l'objecte (node) sobre el qual actua.
- Cal haver afegit l'objecte *Behavior* al Node arrel (*addChild*).

Condicions d'activació

La classe abstracta *WakeupCondition* engloba en una subclasse abstracta *WakeupCriterion* els tipus de condicions d'activació i en quatre classes (*WakeupOr*, *WakeupAnd*, *WakeupAndOfOrs*, *WakeupOrOfAnds*) més les eines necessàries per a fer operacions lògiques entre els tipus de condicions.

Algunes de les condicions de *WakeupCriterion* són:

- *WakeupOnActivation*: Aquesta classe s'activa la primera vegada que la regió de visibilitat de l'escena entra en contacte amb la zona d'influència d'un objecte *Behavior*.
- *WakeupOnAWTEvent*: Aquesta classe s'activa i diferencia els diferents tipus d'esdeveniments *AWT*.
- *WakeupOnElapsedTime*: Aquesta classe s'activa passat un cert temps.
- *WakeupOnTransformChange*: S'activa quan l'objecte associat canvia d'orientació o posició.

Combinacions de condicions

- *WakeupAnd*(*WakeupCriterion[]* conditions): Inicialitza un nou objecte a partir de condicions *WakeupCriterion* i unions AND. Només s'activa quan es donen totes les condicions.
- *WakeupOr*(*WakeupCriterion[]* conditions): Construeix un nou objecte *WakeupOr* a partir de condicions *WakeupCriterion* i unions OR. S'activa quan alguna de les condicions succeix.
- *WakeupAndOfOrs*(*WakeupOr[]* conditions): Construeix un nou objecte a partir de condicions *WakeupOr* i unions AND. S'activa quan succeeixin totes les condicions OR.

- *WakeupOrOfAnds*(*WakeupAnd[]* conditions): Construeix un nou objecte a partir de condicions *WakeupAnd* i unions OR. S'activa quan succeeixi alguna de les condicions AND.

Alguns exemples de classes *Behavior*:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import java.awt.event.*;
import java.util.Enumeration;

//   SimpleBehaviorApp dibuixa un cub
//   que rota quan apretes una tecla.

public class SimpleBehaviorApp extends Applet {

  public class SimpleBehavior extends Behavior{

    private TransformGroup targetTG;
    private Transform3D rotation = new Transform3D();
    private double angle = 0.0;

    SimpleBehavior(TransformGroup targetTG){
      this.targetTG = targetTG;
    }

    public void initialize(){
      this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
    }

    public void processStimulus(Enumeration criteria){
      angle += 0.1;
      rotation.rotY(angle);
      targetTG.setTransform(rotation);
      this.wakeupOn(new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED));
    }

  }

  public BranchGroup createSceneGraph() {
     BranchGroup objRoot = new BranchGroup();

     TransformGroup objRotate = new TransformGroup();
     objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

     objRoot.addChild(objRotate);
     objRotate.addChild(new ColorCube(0.4));

     SimpleBehavior myRotationBehavior = new SimpleBehavior(objRotate);
     myRotationBehavior.setSchedulingBounds(new BoundingSphere());
     objRoot.addChild(myRotationBehavior);

     objRoot.compile();

     return objRoot;
  }

  public SimpleBehaviorApp() {
     setLayout(new BorderLayout());
     GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();

     Canvas3D canvas3D = new Canvas3D(config);
     add("Center", canvas3D);
```

```
        BranchGroup scene = createSceneGraph();

        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);

        simpleU.getViewingPlatform().setNominalViewingTransform();

        simpleU.addBranchGraph(scene);
    }...
...
public class OpenBehavior extends Behavior{

    private TransformGroup  targetTG;
    private WakeupCriterion pairPostCondition;
    private WakeupCriterion wakeupNextFrame;
    private WakeupCriterion AWTEventCondition;
    private Transform3D     t3D = new Transform3D();
    private Matrix3d        rotMat = new Matrix3d();
    private double          doorAngle;

    OpenBehavior(TransformGroup targetTG){
      this.targetTG = targetTG;
      AWTEventCondition = new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED);
      wakeupNextFrame = new WakeupOnElapsedFrames(0);
    }

    public void setBehaviorObjectPartner(Behavior behaviorObject){
      pairPostCondition = new WakeupOnBehaviorPost(behaviorObject, 1);
    }

    public void initialize(){
      this.wakeupOn(AWTEventCondition);
      doorAngle = 0.0;
    }

    public void processStimulus(Enumeration criteria){
      if (criteria.nextElement().equals(pairPostCondition)){
        System.out.println("ready to open door");
        this.wakeupOn(AWTEventCondition);
        doorAngle = 0.0f;
      } else { // could be KeyPress or nextFrame, in either case: open
        if (doorAngle < 1.6){
          doorAngle += 0.1;
          if (doorAngle > 1.6) doorAngle = 1.6;
          // get rotation and scale portion of transform
          targetTG.getTransform(t3D);
          t3D.getRotationScale(rotMat);
          // set y-axis rotation to doorAngle
          // (clobber any previous y-rotation, x and z scale)
          rotMat.m00 = Math.cos(doorAngle);
          rotMat.m22 = rotMat.m00;
          rotMat.m02 = Math.sin(doorAngle);
          rotMat.m20 = -rotMat.m02;
          t3D.setRotation(rotMat);
          targetTG.setTransform(t3D);
          this.wakeupOn(wakeupNextFrame);
        } else { // finished opening door, signal other behavior
          System.out.println("door is open");
          this.wakeupOn(pairPostCondition);
          postId(1);
        }
      }
    }

  }

  public class CloseBehavior extends Behavior{

    private TransformGroup  targetTG;
    private WakeupCriterion pairPostCondition;
    private WakeupCriterion wakeupNextFrame;
    private WakeupCriterion AWTEventCondition;
    private Transform3D     t3D = new Transform3D();
    private Matrix3d        rotMat = new Matrix3d();
    private double          doorAngle;

    CloseBehavior(TransformGroup targetTG){
```

```
        this.targetTG = targetTG;
        AWTEventCondition = new WakeupOnAWTEvent(KeyEvent.KEY_PRESSED);
        wakeupNextFrame = new WakeupOnElapsedFrames(0);
    }

    public void setBehaviorObjectPartner(Behavior behaviorObject){
        pairPostCondition = new WakeupOnBehaviorPost(behaviorObject, 1);
    }

    public void initialize(){
        this.wakeupOn(pairPostCondition);
        doorAngle = 1.6;
    }

    public void processStimulus(Enumeration criteria){
        if (criteria.nextElement().equals(pairPostCondition)){
            System.out.println("ready to close door");
            this.wakeupOn(AWTEventCondition);
            doorAngle = 1.6f;
        } else { // could be KeyPress or nextFrame, in either case: close
            if (doorAngle > 0.0){
                doorAngle -= 0.1;
                if (doorAngle < 0.0) doorAngle = 0.0;
                // get rotation and scale portion of transform
                targetTG.getTransform(t3D);
                t3D.getRotationScale(rotMat);
                // set y-axis rotation to doorAngle
                // (clobber any previous y-rotation, x and z scale)
                rotMat.m00 = Math.cos(doorAngle);
                rotMat.m22 = rotMat.m00;
                rotMat.m02 = Math.sin(doorAngle);
                rotMat.m20 = -rotMat.m02;
                t3D.setRotation(rotMat);
                targetTG.setTransform(t3D);
                this.wakeupOn(wakeupNextFrame);
            } else { // finished opening door, signal other behavior
                System.out.println("door is closed");
                this.wakeupOn(pairPostCondition);
                postId(1);
            }
        }
    }

}...
```

MouseBehavior

El grup de classes encapçalades per *MouseBehavior* serveixen per donar resposta a accions realitzades pel ratolí. Les accions disponibles a les quals dóna resposta són la rotació d'objectes (*MouseRotate*), desplaçaments (*MouseTranslate*) i fer zoom sobre objectes (*MouseZoom* o *MouseWheelZoom*). Totes aquestes classes utilitzen la interfície *MouseBehaviorCallback* (sense implementar-la) que li proporcionen uns mètodes (*transformChanged* i *setupCallback*) que són cridats automàticament per Java3D quan s'ha modificat el node lligat a la classe *Behavior*.

Aplicacions

Una de les aplicacions de les classes *Behavior* és la possibilitat de navegar per l'espai virtual utilitzant el ratolí o les tecles de direcció del teclat.

Navegació amb el ratolí

Per a poder navegar amb el ratolí cal recordar que el node a assignar a la classe *Behavior* ha de ser el que conté el punt de vista de l'observador. Aquest node es pot aconseguir de l'objecte *ViewingPlatform*. Si s'utilitza *SimpleUnivers* es pot aconseguir amb la instrucció següent:

- *simpleU.getViewingPlatform().getViewPlatformTransform()*

I posteriorment utilitzar aquest node (*TransformGroup*) en la creació dels nous objectes *Behavior* (*MouseRotate*, *MouseTranslate* i *MouseZoom*).

Per utilitzar el ratolí per moure's per l'univers virtual cal seguir les següents instruccions.

- Primer cal prémer algun dels botons: esquerre (*MouseRotate*), dret (*MouseTranslate*) o el central (*MouseZoom*).
- Mantenint premut el botó, movem el ratolí.
- El zoom també es pot fer utilitzant la rodeta (*MouseWheelZoom*).

Navegació per teclat

De la mateixa manera que en la navegació amb el ratolí, la navegació amb el teclat és necessari tenir el node del punt de vista de l'observador. Així que una vegada obtingut el node de l'objecte *ViewingPlatform* s'utilitza per assignar-lo a un nou objecte *KeyNavigatorBehavior* i una vegada definida la zona d'influència penjar-lo del node arrel. La utilització pràctica està resumida en el següent quadre:

Un exemple de *MouseBehavior*:

```
public BranchGroup createSceneGraph() {
    BranchGroup objRoot = new BranchGroup();
```

```
TransformGroup objTransform = new TransformGroup();
objTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objTransform.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

objRoot.addChild(objTransform);
objTransform.addChild(new ColorCube(0.4));
objRoot.addChild(new Axis());

MouseRotate myMouseRotate = new MouseRotate();
myMouseRotate.setTransformGroup(objTransform);
myMouseRotate.setSchedulingBounds(new BoundingSphere());
objRoot.addChild(myMouseRotate);

MouseTranslate myMouseTranslate = new MouseTranslate();
myMouseTranslate.setTransformGroup(objTransform);
myMouseTranslate.setSchedulingBounds(new BoundingSphere());
objRoot.addChild(myMouseTranslate);

MouseZoom myMouseZoom = new MouseZoom();
myMouseZoom.setTransformGroup(objTransform);
myMouseZoom.setSchedulingBounds(new BoundingSphere());
objRoot.addChild(myMouseZoom);
...
```

Seleccionar objectes

Classes primitives

El paquet de Java3D, *com.sun.j3d.utils.pickfast*, ens proporciona les eines necessàries per seleccionar objectes del nostre entorn virtual. Ens proporciona les classes *PickTool*, *PickCanvas* i *PickIntersection*.

La primera d'elles defineix els mètodes a utilitzar per a seleccionar objectes de l'espai virtual i moure'ls a voluntat en temps d'execució. *PickCanvas* (subclasse de *PickTool*) s'utilitza per extrapol·lar la posició del punter del ratolí.

El sistema consisteix en projectar un raig des de la selecció en la pantalla cap a l'espai virtual i determinar l'objecte que és selecciona mitjançant la intersecció (*PickIntersection*) amb el raig projectat. *PickTool* utilitza *PickShape* per a definir el tipus de raig.

S'ha de tenir present, a l'hora de permetre seleccionar objectes, la necessitat de tenir activades algunes capacitats per a cada node que es vulgui permetre seleccionar-lo. Les següents capacitats en són un exemple:

- ENABLE_PICK_REPORTING
- ALLOW_BOUNDS_WRITE

- ALLOW_PICKABLE_WRITE

PickShape: Aquesta classe abstracta i les seves subclasses ens permeten definir els tipus de rajos. Segons la classe escollida creen un objecte geomètric amb una certa forma i seleccionen els objectes amb que estigui en contacte. Aquestes són les classes més elementals que té la biblioteca per a seleccionar objectes:

- *PickBounds*: S'utilitza per crear una zona de selecció.
- *PickCone*: Inclou les classes *PickConeRay* i *PickConeSegment* que serveixen per crear rajos en forma de con infinit o finit, respectivament.
- *PickCylinder*: Inclou les classes *PickCylinderRay* i *PickCylinderSegment* que serveixen per crear rajos en forma de cilindre infinit o finit, respectivament.
- *PickPoint*: Serveix per seleccionar un objecte situat en un determinat punt de l'espai.
- *PickRay*: Crea un raig infinit que selecciona tots els objectes que intercepta.
- *PickSegment*: Igual que *PickRay*, excepte que el raig és finit.

Classes d'alt nivell

En la majoria de casos no fa falta utilitzar les classes primitives perquè utilitzant les d'alt nivell ja és suficient. El paquet que les inclou és el *com.sun.j3d.utils.pickfast.behaviors*.

La classe abstracta *PickMouseBehavior* és la superclasse de tres classes que simplifiquen tot el procés de seleccionar i transformar (rotar, moure i fer zoom) sobre els objectes de l'espai virtual.

- *PickRotateBehavior*: Serveix per rotar un objecte, mitjançant el botó esquerre del ratolí.

- *PickTranslateBehavior*: Permet moure de posició un objecte mitjançant el botó dret del ratolí.
- *PickZoomBehavior*: Permet atansar-se o allunyar-se d'un objecte mitjançant el botó central del ratolí.

Les tres classes tenen dos modes de funcionament en el procés de selecció de l'objecte, segons es vulgui rapidesa de càlcul o precisió en la selecció.

- *PickObject*.USE_BOUNDS: Selecciona l'objecte mitjançant una zona esfèrica que l'engloba.
- *PickObject*.USE_GEOMETRY: Selecciona l'objecte mitjançant la seva geometria. Caldrà que tots els objectes de l'escena tinguin activada la capacitat ALLOW_INTERSECT.

Finalment el paquet també conté una interfície, *PickingCallback*, que tota classe interessada en moure objectes amb el ratolí ha d'implementar. L'objecte que es crearà amb aquesta classe estarà lligada amb les subclasses de *PickMouseBehavior* mitjançant el mètode *setupCallback*. Quan un objecte sigui seleccionat es cridarà el mètode *transformChanged* de la interfície *PickingCallback*.

Un exemple que mostra l'ús de *PickRotateBehavior* i que dibuixa dos cubs seleccionables independents:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.behaviors.picking.*;
import javax.media.j3d.*;
import javax.vecmath.*;

import java.awt.event.*;
import java.util.Enumeration;

//   PickCallbackApp renders two interactively rotatable cubes.

public class PickCallbackApp extends Applet {

   public class MyCallbackClass extends Object implements PickingCallback{
      public void transformChanged(int type, TransformGroup tg) {
            System.out.println("picking");
      }
   }

   public BranchGroup createSceneGraph(Canvas3D canvas) {
```

```
        // Create the root of the branch graph
        BranchGroup objRoot = new BranchGroup();

        TransformGroup objRotate = null;
        PickRotateBehavior pickRotate = null;
        Transform3D transform = new Transform3D();
        BoundingSphere behaveBounds = new BoundingSphere();

        // create ColorCube and PickRotateBehavior objects
        transform.setTranslation(new Vector3f(-0.6f, 0.0f, -0.6f));
        objRotate = new TransformGroup(transform);
        objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
        objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);

        objRoot.addChild(objRotate);
        objRotate.addChild(new ColorCube(0.4));

        pickRotate = new PickRotateBehavior(objRoot, canvas, behaveBounds);
        objRoot.addChild(pickRotate);

        PickingCallback myCallback = new MyCallbackClass();
        // Register the class callback to be called
        pickRotate.setupCallback(myCallback);

        // add a second ColorCube object to the scene graph
        transform.setTranslation(new Vector3f( 0.6f, 0.0f, -0.6f));
        objRotate = new TransformGroup(transform);
        objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
        objRotate.setCapability(TransformGroup.ENABLE_PICK_REPORTING);

        objRoot.addChild(objRotate);
        objRotate.addChild(new ColorCube(0.4));

        // Let Java 3D perform optimizations on this scene graph.
        objRoot.compile();

        return objRoot;
    } // end of CreateSceneGraph method of PickCallbackApp

    // Create a simple scene and attach it to the virtual universe

    public PickCallbackApp() {
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
          SimpleUniverse.getPreferredConfiguration();

        Canvas3D canvas3D = new Canvas3D(config);
        add("Center", canvas3D);

        // SimpleUniverse is a Convenience Utility class
        SimpleUniverse simpleU = new SimpleUniverse(canvas3D);

        BranchGroup scene = createSceneGraph(canvas3D);

        // This will move the ViewPlatform back a bit
        simpleU.getViewingPlatform().setNominalViewingTransform();

        simpleU.addBranchGraph(scene);
    } // end of PickCallbackApp (constructor)

    //  The following allows this to be run as an application or as an applet

    public static void main(String[] args) {
        System.out.print("PickCallbackApp.java \n- a demonstration of using the PickRotateBehavior ");
        System.out.println("utility class to provide interaction in a Java 3D scene.");
        System.out.println("Hold the mouse button over a cube then move the mouse to make that cube rotate.");
        System.out.println("This is a simple example progam from The Java 3D API Tutorial.");
        System.out.println("The Java 3D Tutorial is available on the web at:");
        System.out.println("http://java.sun.com/products/java-media/3D/collateral");
        Frame frame = new MainFrame(new PickCallbackApp(), 256, 256);
    } // end of main (method of PickCallbackApp)

} // end of class PickCallbackApp
```

Animacions

Les classes encapçalades per *Interpolator* formen el grup de classes el propòsit de les quals és controlar la resposta de l'aplicació a través del temps, creant animacions. Abans de presentar els diferents interpoladors, és necessari veure com Java3D representa el temps.

Alpha

La classe *Alpha* (Vegeu figura 2.10.)és la forma que té Java3D de representar el temps. Aquesta converteix qualssevol seqüència de temps en un interval de valors entre 0 i 1. La celeritat en passar d'un valor a un altre determina la quantitat de temps.

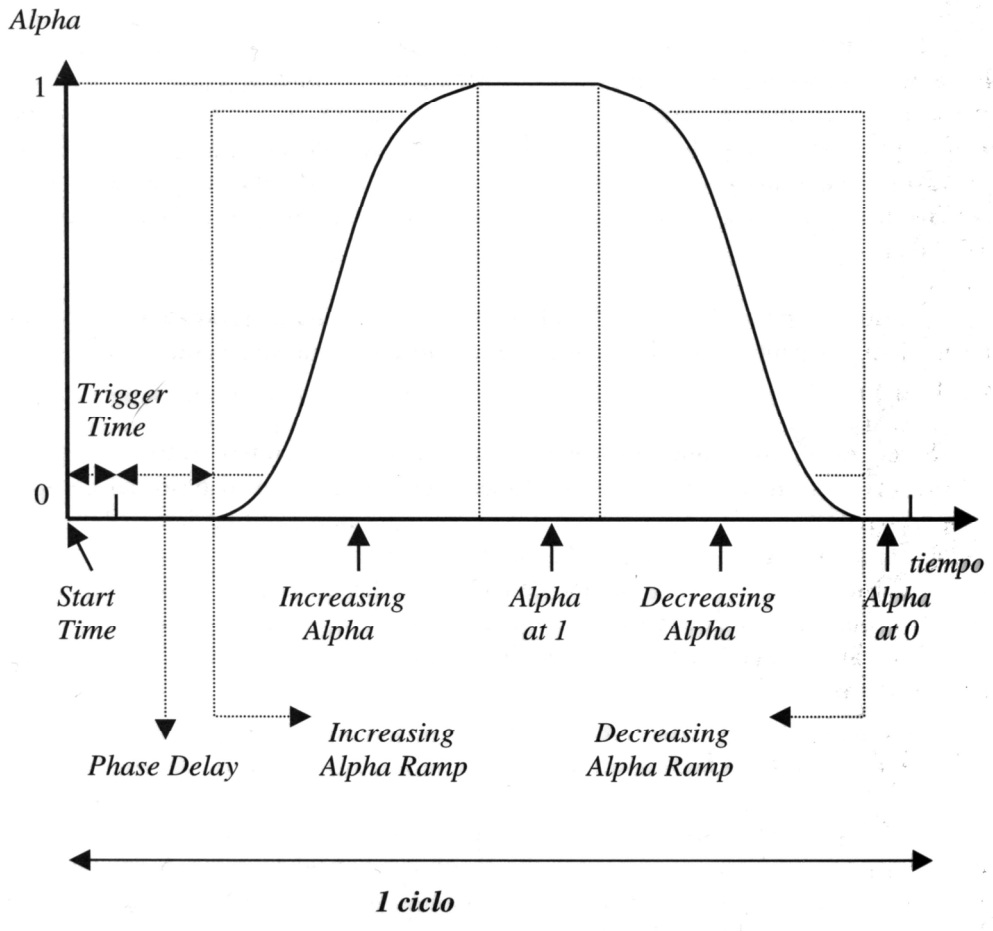

Figura 2.10. Modelització d'un cicle d'*Alpha*

Paràmetres:

startTime: Conté l'hora del rellotge del sistema. El seu valor per defecte és les 0 hores de l'1 de gener de 1970.

triggerTime: Temps que espera *Alpha* a executar-se una vegada carregada l'animació. Valor per defecte 0.

phaseDelayDuration: Temps extra d'espera a començar a executar-se.

increasingAlphaDuration: Temps que tarda l'objecte *Alpha* en passar de 0 a 1.

decreasingAlphaDuration: Temps que tarda l'objecte *Alpha* en passar de 1 a 0.

increasingAlphaRampDuration: Serveix per definir acceleracions entre 0 i 1. Entre 1 i 0 s'utilitza *decreasingAlphaRampDuration*.

loopCount: Determina les vegades que es repetirà l'execució de l'objecte *Alpha*. Valor per defecte -1, que significa que no parará.

mode: Els valors que pot prendre són INCREASING_ENABLE i DECREASING_ENABLE. La utilització del primer valor provocarà que no s'executarà la fase de decreixement, i la utilització del segon la de creixement. La utilització de tots dos farà que s'executi tota la seqüència.

Interpoladors

La classe abstracta *javax.media.j3d.Interpolator* posa les bases i subdivideix els tipus d'interpoladors en diferents grups: una classe interpoladora de colors (*ColorInterpolator*), una de transparències (*TransparencyInterpolator*), una commutadora de nodes (*SwitchValueInterpolator*) i agrupa sota *TransformInterpolator* totes les classes transformadores (rotacions, canvis de

posició i canvis d'escala) de nodes. Els objectes interpoladors obtinguts s'han de penjar d'alguna branca com si fossin un fill (*addchild*).

ColorInterpolator: Serveix per realitzar canvis en l'aparença d'un o varis objectes al llarg del temps. Aquest objecte ha d'estar il·luminat i la seva aparença ha de tenir definit l'objecte *Material*. Aquest és l'objecte sobre el qual s'aplica l'objecte *ColorInterpolator*, i com que és un *NodeComponent* pot estar lligat a diverses aparences i així l'interpolador afecti a diversos objectes al mateix temps. Aquests canvis en l'objecte *Material* afecten a l'atribut *Diffuse* i segons el constructor que s'utilitzi determinarà el seu comportament.

- *ColorInterpolator*(*Alpha* alpha, *Material* material): Transformarà l'atribut *Diffuse* de negre a blanc.
- *ColorInterpolator*(*Alpha* alpha, *Material* material, *Color3f* colorini, *Color3f* colorfin): Transformarà l'atribut *Diffuse* interpolant entre els colors donats.

TransparencyInterpolator: Aquesta classe com l'anterior, no s'aplica sobre un *Node* sinó sobre l'atribut (*NodeComponent*) de transparència d'una aparença (*Appareance*). L'efecte d'aquesta classe és interpolar entre dos valors de transparència.

SwitchValueInterpolator: La funció d'aquesta és la de commutar entre diferents Nodes. Aquests han d'afegir-se a un objecte *Switch* com si fossin fills (*addchild*). Indicant-li quin és el primer i l'últim dels fills de l'objecte *Switch*, en la creació de l'objecte *SwitchValueInterpolator*, aquest reparteix els Nodes per tot *Alpha* a distàncies equidistants. Llavors amb el pas del temps (*Alpha*) van apareixen i desapareixen les rames que l'objecte *Switch* té com a fills.

TransformInterpolator: Les subclasses d'aquesta classe abstracta engloben els diversos de transformacions de posició, rotacions i canvis d'escala (a més d'alguna classe per casos particulars); apart d'un grup que inclou les mateixes transformacions, però tractades d'un forma diferent. Les transformacions són:

-*PositionInterpolator*: Aquesta classe modifica un Node entre dues posicions, inicial i final, creant una interpolació del Node durant el pas del temps.

-*RotationInterpolator*: L'efecte que provoca aquesta classe és la interpolació del Node seleccionat entre dos angles; creant una rotació del Node en un eix de gir determinat, per defecte l'eix Y.

-*ScaleInterpolator*: Aquest interpolador s'encarrega de modificar la mida de tots els objectes del Node seleccionat per canvi d'escala i els interpola linealment entre les dues escales donades.

-*PathInterpolator*: És la classe abstracta que encapçala una sèrie de classes emprades per modificar Nodes aplicant-li modificacions iguals als tres últims interpoladors; però amb una diferència, la introducció dels pesos (*knots*). El resultat de la utilització de les subclasses de *PathInterpolator* dependrà del pas del temps (*Alpha*) i uns pesos, anomenats *knots*. La utilització d'aquests pesos permet la definició de diferents transformacions consecutives i assignar un pes a cada una d'elles i així repartir les transformacions per *Alpha*. Per definir les transformacions s'utilitzen quaternions.

-*RotationPathInterpolator*: Aquesta classe modifica el Node indicat creant rotacions definides per una cadena de quaternions *Quad4f* i els pesos, *knots*, carregats en el constructor.

-*RotPosPathInterpolator*: Utilitza una cadena de quaternions *Quad4f* per a les rotacions i una altra cadena de punts, *Point3f*, per a les posicions intermèdies. Els pesos determinaran el temps gastat en cada interpolació.

-*RotPosScalePathInterpolator*: Utilitza una cadena de quaternions *Quad4f* per a les rotacions, una altra cadena de punts, *Point3f*, per a les posicions intermèdies i una cadena de *float* per a les diferents escales que han de prendre els objectes del Node assignat. Els pesos determinaran el temps gastat en cada interpolació.

-*PositionPathInterpolator*: Utilitza una cadena de punts, *Point3f*, per a les posicions intermèdies d'una sèrie de canvis de posició. Els pesos determinaran el temps gastat en cada interpolació.

Un exemple on mostra la classe *Alpha* i diversos interpoladors:

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.ColorCube;
import javax.media.j3d.*;
import javax.vecmath.*;

public class InterpolatorApp extends Applet {

   Shape3D createCar(float xScale, float yScale, boolean createNormals, boolean assignColoring) {

      Shape3D car = new Shape3D();
      QuadArray carGeom = null;

      if (createNormals)
         carGeom = new QuadArray(16, GeometryArray.COORDINATES
                           | GeometryArray.NORMALS);
      else
         carGeom = new QuadArray(16, GeometryArray.COORDINATES);

      carGeom.setCoordinate( 0, new Point3f(xScale*-0.25f, yScale*0.22f, 0.0f));
      carGeom.setCoordinate( 1, new Point3f(xScale* 0.20f, yScale*0.22f, 0.0f));
      carGeom.setCoordinate( 2, new Point3f(xScale* 0.10f, yScale*0.35f, 0.0f));
      carGeom.setCoordinate( 3, new Point3f(xScale*-0.20f, yScale*0.35f, 0.0f));
      carGeom.setCoordinate( 4, new Point3f(xScale*-0.50f, yScale*0.10f, 0.0f));
      carGeom.setCoordinate( 5, new Point3f(xScale* 0.50f, yScale*0.10f, 0.0f));
      carGeom.setCoordinate( 6, new Point3f(xScale* 0.45f, yScale*0.20f, 0.0f));
      carGeom.setCoordinate( 7, new Point3f(xScale*-0.48f, yScale*0.20f, 0.0f));
      carGeom.setCoordinate( 8, new Point3f(xScale*-0.26f, yScale*0.00f, 0.0f));
      carGeom.setCoordinate( 9, new Point3f(xScale*-0.18f, yScale*0.00f, 0.0f));
      carGeom.setCoordinate(10, new Point3f(xScale*-0.16f, yScale*0.12f, 0.0f));
      carGeom.setCoordinate(11, new Point3f(xScale*-0.28f, yScale*0.12f, 0.0f));
      carGeom.setCoordinate(12, new Point3f(xScale* 0.25f, yScale*0.00f, 0.0f));
      carGeom.setCoordinate(13, new Point3f(xScale* 0.33f, yScale*0.00f, 0.0f));
      carGeom.setCoordinate(14, new Point3f(xScale* 0.35f, yScale*0.12f, 0.0f));
      carGeom.setCoordinate(15, new Point3f(xScale* 0.23f, yScale*0.12f, 0.0f));

      if (createNormals){
         int i;
         Vector3f normal = new Vector3f(0.6f, 0.6f, 0.8f);
         for(i = 0; i < 8; i++)
            carGeom.setNormal(i, normal);
         normal.set(new Vector3f(0.5f, 0.5f, 0.5f));
         for(i = 8; i <16; i++)
            carGeom.setNormal(i, normal);
      }

      if (assignColoring){
```

```
        ColoringAttributes colorAttrib =
            new ColoringAttributes(0.0f, 0.0f, 1.0f, ColoringAttributes.NICEST);
        Appearance carAppear = new Appearance();
        carAppear.setColoringAttributes(colorAttrib);
        car.setAppearance(carAppear);
    }

    car.setGeometry(carGeom);
    return car;
}

public BranchGroup createSceneGraph() {
        // Create the root of the branch graph
        BranchGroup objRoot = new BranchGroup();
    Transform3D t3d = new Transform3D();
    BoundingSphere bounds = new BoundingSphere();

    // create target TransformGroup with Capabilities
    TransformGroup objMove = new TransformGroup();
    objMove.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

    // create target TransformGroup with Capabilities
    TransformGroup objRotate = new TransformGroup();
    objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

    // create target TransformGroup with Capabilities
    TransformGroup objScale = new TransformGroup();
    objScale.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

    // create target Material with Capabilities
    Material objColor = new Material();
    objColor.setCapability(Material.ALLOW_COMPONENT_WRITE);

    // create target Transparency with Capabilities
    TransparencyAttributes objTransp = new TransparencyAttributes();
    objTransp.setCapability(TransparencyAttributes.ALLOW_VALUE_WRITE);
    objTransp.setTransparencyMode(TransparencyAttributes.BLENDED);

    // create target Switch with Capabilities
    Switch objSwitch = new Switch();
    objSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);

    // create Alpha
    Alpha alpha = new Alpha (-1,
                    Alpha.INCREASING_ENABLE + Alpha.DECREASING_ENABLE,
                    0, 0, 2000, 0, 1000, 2000, 0, 1000);

    // create position interpolator
    PositionInterpolator posInt = new PositionInterpolator (alpha, objMove);
    posInt.setSchedulingBounds(bounds);
    posInt.setStartPosition(-1.0f);

    // create rotation interpolator
    RotationInterpolator rotInt = new RotationInterpolator (alpha, objRotate);
    rotInt.setSchedulingBounds(bounds);

    // create scale interpolator
    ScaleInterpolator scaInt = new ScaleInterpolator (alpha, objScale);
    scaInt.setSchedulingBounds(bounds);

    // create color interpolator
    ColorInterpolator colInt = new ColorInterpolator (alpha, objColor);
    colInt.setStartColor(new Color3f(1.0f, 0.0f, 0.0f));
    colInt.setEndColor(new Color3f(0.0f, 0.0f, 1.0f));
    colInt.setSchedulingBounds(bounds);

    // create transparency interpolator
    TransparencyInterpolator traInt = new TransparencyInterpolator (alpha, objTransp);
    traInt.setSchedulingBounds(bounds);

    // create switch value interpolator
    SwitchValueInterpolator swiInt = new SwitchValueInterpolator (alpha, objSwitch);
    swiInt.setSchedulingBounds(bounds);

    t3d.setTranslation(new Vector3f(0.0f, 0.8f, 0.0f));
    TransformGroup objMovePos = new TransformGroup(t3d);
```

```
objRoot.addChild(objMovePos);
objMovePos.addChild(objMove);
objMove.addChild(createCar(0.4f, 0.4f, false, true));
objRoot.addChild(posInt);

t3d.setTranslation(new Vector3f(0.0f, 0.5f, 0.0f));
TransformGroup objRotPos = new TransformGroup(t3d);
objRoot.addChild(objRotPos);
objRotPos.addChild(objRotate);
objRotate.addChild(createCar(0.4f, 0.4f, false, true));
objRoot.addChild(rotInt);

t3d.setTranslation(new Vector3f(0.0f, 0.2f, 0.0f));
TransformGroup objScalePos = new TransformGroup(t3d);
objRoot.addChild(objScalePos);
objScalePos.addChild(objScale);
objScale.addChild(createCar(0.4f, 0.4f, false, true));
objRoot.addChild(scaInt);

t3d.setTranslation(new Vector3f(0.0f, -0.2f, 0.0f));
TransformGroup objColorPos = new TransformGroup(t3d);
objRoot.addChild(objColorPos);
Shape3D colorCar = createCar(0.4f, 0.4f, true, false);
Appearance materialAppear = new Appearance();
materialAppear.setMaterial(objColor);
colorCar.setAppearance(materialAppear);
objColorPos.addChild(colorCar);
objRoot.addChild(colInt);

t3d.setTranslation(new Vector3f(0.0f, -0.5f, 0.0f));
TransformGroup objTranspPos = new TransformGroup(t3d);
objRoot.addChild(objTranspPos);
Shape3D transpCar = createCar(0.4f, 0.4f, false, true);
Appearance transpAppear = transpCar.getAppearance();
transpAppear.setTransparencyAttributes(objTransp);
objTranspPos.addChild(transpCar);
objRoot.addChild(traInt);

t3d.setTranslation(new Vector3f(0.0f, -0.8f, 0.0f));
TransformGroup objSwitchPos = new TransformGroup(t3d);
objRoot.addChild(objSwitchPos);
objSwitch.addChild(createCar(0.4f, 0.4f, false, true));;
objSwitch.addChild(new ColorCube(0.1f));
objSwitchPos.addChild(objSwitch);
objRoot.addChild(swiInt);
swiInt.setLastChildIndex(2);// since switch made after interpolator
    ...
```

Morph

En un nivell molt més avançat en el tema de les animacions, Java3D proporciona una classe anomenada *Morph*. Mentre que les classes interpoladores fan canvis d'aparença o transformacions de posició, rotació o d'escala; la classe *Morph* basa la seva funcionalitat en transformacions geomètriques. Per si sola aquesta classe no produeix can animació, però defineix les geometries intermèdies entre dos objectes, un d'inicial i un de final. També emmagatzema el pes (semblant als *knots*) o valor de cada una d'aquestes geometries. L'animació s'aconsegueix mitjançant el disseny d'una classe *Behavior*. El següent codi exemplifica una classe *Behavior*:

```
public class MorphBehavior extends Behavior {

    Alpha alpha;
    Morph morph;
    double weights[];
    WakeupOnElapsedFrames w = new WakeupOnElapsedFrames(0);

    public void initialize() {
           alpha.setStartTime(System.currentTimeMillis());
           wakeupOn(w);
    }

    public void processStimulus(Enumeration criteria) {

           double val = alpha.value();
           if (val < 0.5) {
               double a = val * 2.0;
               weights[0] = 1.0 - a;
               weights[1] = a;
               weights[2] = 0.0;
           }
           else {
               double a = (val - 0.5) * 2.0;
               weights[0] = 0.0;
               weights[1] = 1.0f - a;
               weights[2] = a;
           }
           morph.setWeights(weights);
           wakeupOn(w);
    }

    public MorphBehavior(Alpha a, Morph m) {
           alpha = a;
           morph = m;
           weights = morph.getWeights();
    }
}
```

Aquest exemple necessita d'un objecte *Morph* i d'un d'*Alpha* per poder-se emprar. La classe que vulgui crear una animació cal que segueixi els següents passos:

- Posar les geometries inicial, final i intermèdies en un *array*.
- Crear l'objecte *Morph* amb l'*array* i definint una aparença.
- Activar les capacitats *Morph*.ALLOW_WEIGHTS_READ i ALLOW_WEIGHTS_WRITE per poder accedir als pesos (utilitzant els mètodes set/get)
- Afegir l'objecte *Morph* a la Branca de contingut.
- Crear l'objecte *Alpha*.
- Crear l'objecte *Behavior* amb els objectes *Morph* i *Alpha*.
- Afegir l'objecte *Behavior* a la Branca de contingut.

2.3. Tecnologia: JMathPlot

Que és JMathPlot?

JMathPlot és una eina que serveix per a dibuixar tota mena de diagrames i permet un alt grau d'interactivitat, així com un *plotter* interactiu. Aquesta eina forma part d'una biblioteca més gran anomenada *JMathTools*, dissenyada i implementada en Java2, sota llicència *BSD*, per Yann RICHET (Vegeu [11]).

JMathPlot permet crear tot tipus de diagrames, de punts, de barres, de línies, de caixa i alguns d'altres. També incorpora una barra amb eines, algunes serveixen per fer zoom a determinades zones del diagrama, definir els eixos o modificar o definir les dades a representar.

Com està estructurada?

JMathPlot està estructurada com l'API de Java2 (Vegeu [13]) i com la majoria de llibreries de Java2, en forma de paquets i sub-paquets. Tota la biblioteca penja de *org.math*; en aquest punt es divideix en dos paquets anomenats *io* i *plot*. Cada un d'ells engloba una funcionalitat que ve determinada per l'objectiu de les classes que inclou.

El paquet *io* inclou totes les classes necessàries per a l'entrada i sortida de dades, és a dir, classes per a carregar o crear fitxers (paquet *files*), classes per llegir o escriure aquests fitxers (binaris) mitjançant fluxos (paquet *stream*) i classes per interpretar aquests fitxers (paquet *parser* amb la seva única classe *ArrayString*). També inclou un paquet per a controlar el flux i donar-li un ordre *LittleIndian* (paquet *littleendian*). A més a més incorpora unes interfícies per a poder imprimir des del fitxer, des d'un *String* o del Portapapers.

El paquet *plot* és el nucli de la biblioteca i està subdividit en diferents subpaquets: *canvas, components, icons, plotObjects, plots, render* i *utils*. En un primer moment ens trobem les classes amb que ens comunicarem amb la biblioteca com són *PlotPanel,* les seves subclasses *Plot2DPanel*, per a gràfiques en 2D, i *Plot3DPanel*, per a 3D. Aquests són panells on succeirà l'aplicació, així com en uns altres panells auxiliars per canviar paràmetres o entrar dades que són: *DataPanel*, *MatrixTablePanel* o *ParametersPanel*. També inclou una

classe que permet executar la biblioteca com una aplicació independent i en un *JFrame* enlloc de fer-ho com a *JPanel* formant part d'un *applet*.

El subpaquet *canvas* inclou la classe abstracta *PlotCanvas* i les seves subclasses *Plot3DCanvas* i *Plot2DCanvas*. Aquestes són el panell sobre el qual es dibuixaran les gràfiques i fa de contenidor de tot el que es dibuixa.

El subpaquet *components* inclou les classes del contingut dels panells: la barra d'eines del panell de dades, *DataToolBar*; la barra d'eines del panell principal, *PlotToolBar*; o el panell de la llegenda, *LegendPanel*, i la pròpia classe llegenda, *Legend*, inclosa dins *LegendPanel*. També hi ha classes per a representar el panell de dades quan s'executi com a *JFrame*, anomenada *DatasFrame* i una altra per a canviar l'escala de la gràfica anomenada *SetScalesFrame*.

El subpaquet *plotObjects* conté els objectes unitaris dels objectes que poden ser dibuixats i les característiques que poden tenir aquests objectes. Els objectes són els eixos de coordenades, *Axe*; els noms, les unitats o en conjunt els caràcters que poden aparèixer, *Label*; les línies de qualssevol tipus, *Line*; o l'aparença de la base de la gràfica i la que conté les referències al sistema de coordenades (*Base*) i als eixos, *BasePlot*. Les característiques que poden prendre les classes descrites són un conjunt d'interfícies: *Editable*, *Plotable*, *BaseDependant* i *Noteable*. La primera d'elles serveix per a poder editar o canviar les dades d'una gràfica mentre s'està executant; *Plotable* defineix l'objecte d'una classe com a un objecte que es pot dibuixar (tots el tipus de gràfiques); *BaseDependant* marca les classes que la seva definició i/o representació depenen de la *Base* sobre la qual estan definides (com són les classes *Axe* o *BasePlot*). *Noteable* dona la possibilitat de destacar l'objecte per sobre els altres amb un color.

El subpaquet *plots* inclou tots els tipus de gràfiques que es poden representar amb la biblioteca, és a dir, Digrames de capsa (*BoxPlot2D* i *BoxPlot3D*), histogrames (*HistogramPlot2D* i *HistogramPlot3D*),

digrames de línia (*LinePlot*) o de barres (*BarPlot*), entre d'altres. Totes aquestes classes són filles d'una classe abstracta anomenada *Plot*, també inclosa dins el paquet.

El subpaquet *render* conté les classes dibuixants, és a dir, les classes que dibuixen les gràfiques, els eixos, etc. El paquet ve encapçalat per la classe abstracta *AbstractDrawer* seguida de *AWTDrawer* i de les seves dues classes *AWTDrawer2D* i *AWTDrawer3D*. També hi ha incloses una classe abstracta anomenada *Projection* i les seves classes hereves *Projection2D* i *Projection3D*, utilitzades per coordinar-se amb els eixos de coordenades.

Finalment el subpaquet *utils* conté dues classes anomenades *Array* i *Histogram*. La primera d'elles, *Array*, conté una sèrie de mètodes estàtics de tractament d'*arrays* de *double* personalitzat per a la biblioteca. Aquests *arrays* és la forma que té la biblioteca de guardar les dades de les gràfiques. La classe *Histogram* conté, com en l'anterior classe, una sèrie de mètodes estàtics que serveixen per manipular les dades d'un histograma i poder definir l'escala dels eixos.

Ús de JMathPlot

Per a utilitzar aquesta eina cal fer-ho des d'un entorn *SWING* ja que la classe abstracta inicial (*PlotPanel*) és una subclasse de *JPanel* i depenent del tipus de gràfiques que vulguis fer, cal utilitzar la classe *Plot2DPanel* o bé *Plot3DPanel*.

Constructors

Són varis els constructors que proporcionen aquestes classes, alguns d'ells:

public PlotPanel(PlotCanvas _canvas, String legendOrientation)

Aquest és el constructor que es crida, normalment, en darrera instància. El paràmetre _canvas fa referència al suport on es dibuixarà les gràfiques i *legendOrientation* és la posició (EAST, NORTH, WEST, SOUTH, INVISIBLE) on es vol posar la llegenda de les gràfiques. Aquest constructor prepara el

Canvas per a la introducció de gràfiques. No és, habitualment, un constructor que l'usuari cridi directament.

public PlotPanel(PlotCanvas _canvas)

Aquest és un constructor que es crida quan no es vol llegenda, per què internament, crida l'anterior constructor amb el paràmetre INVISIBLE.

public Plot2DPanel(double[] min, double[] max, String[] axesScales, String[] axesLabels)

Aquest és un constructor molt útil que permet definir els valors mínims i els màxims que tindran els eixos i el nom a posar als eixos. Internament crida l'anterior constructor amb una crida personalitzada amb els paràmetres al constructor de *Plot2DCanvas* (o al 3D).

Mètodes

La resta de mètodes que contenen les classes *PlotPanel* i les seves hereves vénen diferenciats precisament pels definits a *PlotPanel* i les pròpies de cada una de les seves subclasses.

Els mètodes definits a *PlotPanel* inclouen mètodes per afegir, treure i col·locar la llegenda i la barra d'eines:

public void addLegend(String o): afegir llegenda
public void removeLegend(): treure llegenda
public void setLegendOrientation(String o): orientació de la llegenda
public void addPlotToolBar(String o): afegir barra d'eines
public void removePlotToolBar (): treure barra d'eines
public void setPlotToolBarOrientation(String o)
(les opcions per la *String* són EAST, NORTH, WEST, SOUTH, INVISIBLE)

Uns mètodes set/get per definir accions a realitzar:

public void setActionMode(int am): referent al comportament dels events de ratolí. Valors: ZOOM(0) o TRASLATION(1)
public void setNoteCoords(boolean b): Activar les coordenades de la *Note*
public void setEditable(boolean b): definir el *canvas* com a editable o no
public boolean getEditable(): saber si està activat
public void setNotable(boolean b): activar o desactivar la *Note*
public boolean getNotable()

Uns mètodes *set/get* per tractar amb els components del *canvas*, com poden ser les diferents gràfiques (*plots*), eixos, noms dels eixos i escala dels eixos; alguns d'ells:

public Plot[] getPlots(): retorna tots els *plots*
public Plot getPlot(int i): retorna el *plot* de l'índex i
public int getPlotIndex(Plot p): retorna l'índex del *plot* p
public Axe getAxe(int i): retorna l'eix de coordenada de l'índex i

public String[] getAxesScales(): retorna totes les escales dels eixos
public void setAxesLabels(String... labels): determina els noms dels eixos
public void setFixedBounds(double[] min, double[] max): determina el valor màxim i el mínim dels eixos
public void setAutoBounds(): determina que el programa calculi els valors màxims dels eixos mitjançant les dades

I finalment uns mètodes per afegir i esborrar dades:

public void addLabel(String text, Color c, double... where): utilitzat per afegir un text en una determinada posició
public int addPlot(Plot newPlot): afegeix la gràfica *newPlot*
public void changePlotData(int I, double[]... XY): canvia les dades de la gràfica determinada per *I*
public void removePlot(int I): esborra la gràfica *I*
public void addQuantiletoPlot(int numPlot, double[]... q): afegeix les quantil·les a una gràfica deteminada *I*
public int addLinePlot(string name, double[]... XY): aquest és el mètode principal per afegir diagrames, en aquest cas afegeix un digrama de línies. Substituint ***Line*** per ***Bar, Box, Scatter, Histogram*****n** o ***Grid*** s'obtenen els diferents tipus de diagrames.

3. Aplicacions desenvolupades: la llei de Faraday-Lenz d'inducció magnètica

3.1. Fonaments físics

Flux magnètic

El flux magnètic és la magnitud matemàtica relacionada amb el nombre de línies camp magnètic que travessen una superfície. (Vegeu [3], [4] i [5])

Donat l'àrea d'una superfície *d*A i el vector **n** unitari Normal a aquesta superfície, el flux magnètic es defineix per l'expressió:

$$\Phi_m = \oint_S B \cdot n \cdot \partial A = \oint_S B_n \partial A$$

La unitat del flux magnètic és la del camp magnètic (tesla) per la unitat d'àrea (metre quadrat), i se l'anomena weber (Wb).

En el cas que la superfície d'àrea A i el camp magnètic B siguin constants, i l'angle (θ) format per la direcció del camp magnètic i el vector Normal sigui diferent de O (Vegeu figura 3.1), el flux queda:

$$\Phi_m = B \cdot A \cdot \cos\theta$$

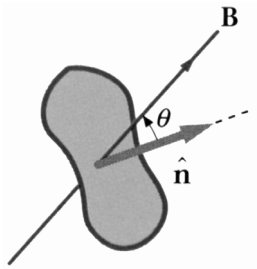

Figura 3.1. Angle θ entre vector Normal i vector camp magnètic

En el cas que l'àrea A formi part d'una de les voltes d'una bobina el resultat del flux es multiplicaria pel nombre de voltes. (Vegeu figura 3.2)

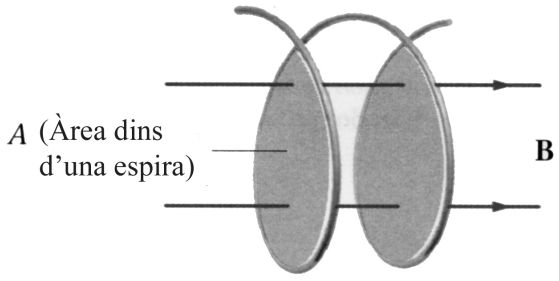

Figura 3.2. Espira amb 2 voltes

Força electromotriu induïda. Llei de Faraday

Els experiments de Faraday (Vegeu [3], [4] i [5]) i altres van demostrar que si el flux

magnètic a través d'una àrea rodejada per un circuit varia per qualsevol motiu, s'indueix una força electromotriu que és igual en magnitud a la variació en el temps del flux. (Vegeu [3], [4] i [5])
Si utilitzem una espira d'un conductor en un camp magnètic, com en la figura. Si el flux és variable s'hi indueix una força electromotriu. Donat que és un conductor s'hi indueix un camp elèctric distribuït per tot el conductor. La força electromotriu per definició és la integral lineal del camp elèctric al voltant de tot el circuit, és a dir:

$$\varepsilon = \oint_C E \cdot \partial \ell$$

Però donat que aquest camp elèctric prové d'una variació de flux magnètic no conservatiu, aquesta integral és igual a la força electromotriu induïda, la qual és igual a la variació en el temps del flux magnètic:

$$\varepsilon = \oint_C E \cdot \partial \ell = -\frac{\partial \Phi_m}{\partial t}$$

Llei de Lenz

El signe negatiu de la llei de Faraday està relacionat amb la direcció de la força electromotriu induïda. (Vegeu [3], [4] i [5]) La direcció i el sentit de la força electromotriu i el corrent induïts poden determinar-se mitjançant el principi físic anomenat Llei de Lenz:

> *La força i la corrent induïts tenen una direcció i sentit tal que tendeixen a oposar-se a la variació que les produeix.*

3.2. Applet 1: la llei de Lenz en un sistema imant-espira

3.2.1 Objectius

- Crear una simulació qualitativa, mitjançant la biblioteca *Fislets* (Vegeu [1] i [2]), del procés d'inducció en un sistema imant-espira, produït pel moviment de l'imant al llarg de l'eix de l'espira.
- Il·lustrar el significat físic de la Llei de Lenz dins de la Llei de Faraday (Vegeu [3], [4] i [5]). És a dir, el significat del signe negatiu en la llei de Faraday.
- Proporcionar als usuaris de l'*applet* paràmetres amb que interactuar amb l'*applet* i que amb la seva modificació visualitzar l'efecte que tenen en el corrent induït a l'espira. Aquests paràmetres són el sentit del moviment de l'imant i la velocitat d'aquest.

La intenció d'aquest *applet* és que l'usuari de l'aplicació pugui veure que ocorre en l'espira quan es mou l'imant, quines diferències apareixen quan es mou cap a la dreta o quan es mou cap a l'esquerra; també quines són les diferències quan es mou lluny o a prop de l'espira i que passa quan s'atura l'imant.

En el cas que l'imant es dirigeixi cap a la dreta (vegeu figura 3.3) fa augmentar el flux que travessa l'espira. Aquesta variació de flux provoca una corrent elèctrica induïda en l'interior de l'espira que al seu temps provoca un camp magnètic induït que s'oposa al de l'imant, és a dir, cap a l'esquerra. En el cas que l'imant vagi cap a l'esquerra, el camp magnètic induït anirà cap a la dreta.

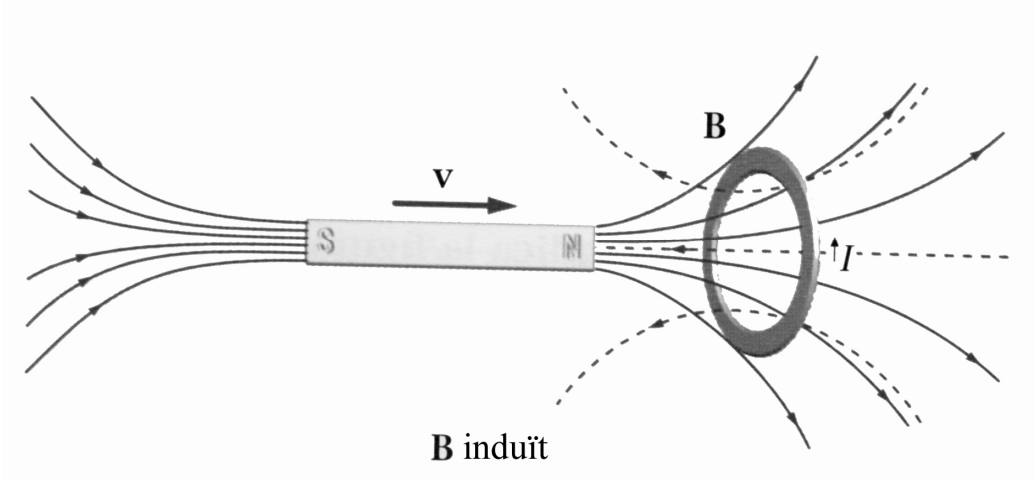

Figura 3.3. Sistema imant-espira

3.2.2. Disseny i implementació

En el disseny d'aquest *applet* vaig guiar-me en els que incorpora la biblioteca. Vaig col·locar l'animació en l'*applet* i les opcions i paràmetres interactius en la pàgina *web* mitjançant *html* i *JavaScript*.
El programa es compon d'un *applet* i d'una sèrie d'opcions en forma de botons normals i botons tipus "radio". L'activació dels quals modifica el que es representa en l'*applet*.
La implementació d'aquest *applet* ha estat mitjançant la biblioteca *Physlets* que utilitza unes classes *applet* predefinides i l'usuari interacciona amb elles, mitjançant llenguatge *JavaScript* i *html*.

JavaScript

La part programada en *JavaScript* es posa dintre de la part d'una pàgina *web* anomenada capçalera (*head*) i entre les clàusules "*script*".

La construcció de l'imant es fa mitjançant l'agrupació de càrregues positives i negatives en dues fileres per a donar l'efecte, en les línies de camp magnètic, d'un imant. L'imant es crea en la funció *posarimant()* que es crida en la funció *initPage()*, que s'executa al carregar l'*applet*.

També en aquesta funció, *initPage()*, es criden les funcions *carrfotos()* i *carrinduit()* que s'encarreguen respectivament d'afegir totes les imatges necessàries per a representar l'espira i les fletxes que donen el sentit al corrent elèctric i al camp magnètic induït produït per aquest, i de col·locar dues càrregues en els extrems de l'espira que simularan aquest camp.

Les funcions *dreta()* i *esquerra()* són les que creen les simulacions que mouen l'imant cap a la dreta i cap a l'esquerra, al mateix temps que modifiquen la intensitat del camp magnètic induït depenent de la distància de l'imant a l'espira. Per a controlar l'animació s'utilitza el rellotge que incorporen aquests *applets*, és a dir que per a simular l'animació es canvia la posició de l'imant mentre el rellotge està en marxa.
Dintre d'aquestes funcions es criden les funcions *fotodreta()* i *fotoesquerra()* que controlen quines imatges (fletxes i espira) que es veuen en cada posició.

Html

En la part del cos (*body*) d'una pàgina *web* hi ha dos parts diferenciades que són: la clàusula "*applet*" i els formularis (*form*).

La clàusula "*applet*" engloba la definició del *applet*, dels arxius necessaris i la seva localització i la definició dels paràmetres inicials.
En la variable *codebase* si defineix la localització dels arxius de les classes Java a partir de la localització de l'arxiu *html*.
La variable *archive* serveix per indicar els arxius a utilitzar. I finalment la variable *code* indica la ruta fins la classe que es crida a l'iniciar l'*applet*.

Dins de la clàusula *applet* també s'inclou un conjunt de paràmetres, cada un dins una clàusula *param*, on s'hi defineix el nom del paràmetre (*name*) i el seu valor (*value*).

Els formularis que estan entre clàusules *form* s'utilitzen per a la creació d'opcions que modifiquen l'*applet* i també per a definir el seu comportament mitjançant la crida, en una variable anomenada *onclick*, a una funció de l'apartat *JavaScript*.
Així doncs el primer formulari serveix per amagar o mostrar el camp magnètic i el camp induït. El segon per fer moure o para l'imant i el tercer per modificar la velocitat d'aquest moviment.

3.2.3. Instal·lació i ús

Per a poder visualitzar aquest *applet* cal mantenir l'estructura de directoris intacta, és a dir, els directoris *applets* i *II4Electromagnetismo* han de ser al mateix directori. Si aquesta estructura es vol modificar cal actualitzar la variable *codebase* dintre de la clàusula *applet* de l'arxiu *html*, i posar-hi la ruta des de l'arxiu *html* fins als arxius de classes.
A més a més cal utilitzar un navegador que tingui instal·lat el *plugin* de Java i que permeti l'execució de llenguatge *JavaScript*.

La manera d'utilitzar aquest *applet* és molt senzilla, els mateixos botons indiquen el que fan. N'hi ha uns que serveixen per mostrar o amagar el camp magnètic i l'induït, els més importants que serveixen per moure l'imant o aturar-lo i finalment uns per modificar la velocitat de moviment de l'imant.

La utilització dels botons "cap a la dreta" i "cap a l'esquerra" provoquen el moviment de l'imant cap a la direcció esmentada. Això crea una variació de flux magnètic en l'interior de l'espira. Aquesta variació indueix un corrent elèctric en l'espira que a la vegada crea una força electromotriu. La direcció i el sentit del corrent i de la força electromotriu és sempre contraria a la direcció i el sentit de la variació que els provoca, és a dir, la variació de flux. El moviment de l'imant fa aparèixer en l'espira de la dreta el corrent i la força induïdes en forma de fletxes que s'oposen al flux.

La utilització dels botons per modificar la velocitat de l'imant fa que a l'iniciar el moviment l'imant es mogui més ràpidament. Això provoca una major variació de flux i per tant una major intensitat de corrent elèctric en l'espira i força electromotriu induïda.

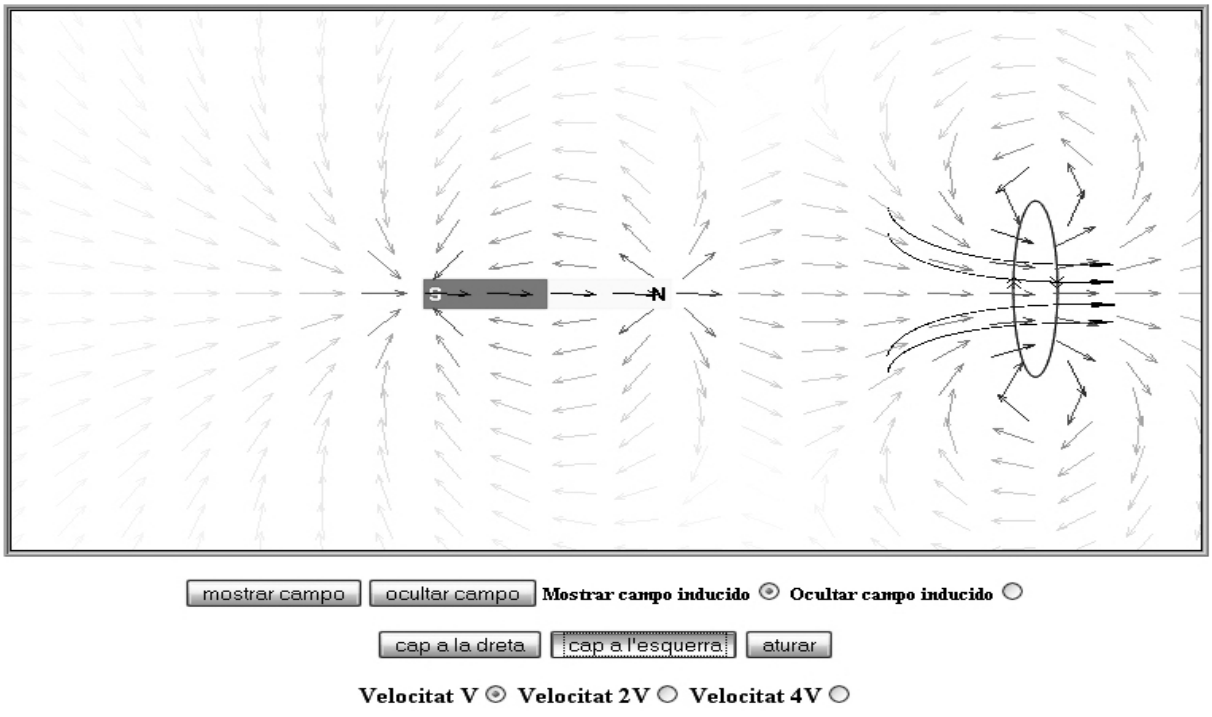

Figura 3.4. Applet 1 en moviment "cap a l'esquerra"

3.3. Applet 2: moviment d'una espira a l'interior d'un camp magnètic uniforme

3.3.1. Objectius

- Un dels objectius d'aquest *applet* és aconseguir que l'usuari comprengui quins són els efectes de la rotació de l'espira entre els dos pols d'un imant i quines són les forces que s'hi generen.
- L'objectiu de la segona animació és que l'usuari comprengui que és una vista de perfil de l'espira on es pot apreciar el seu vector Normal i la direcció del camp magnètic. Amb el moviment de rotació s'aconsegueix la variació de l'angle entre el vector Normal i el vector director del camp magnètic, i amb això la variació de flux magnètic.
- Els resultats d'experimentar amb l'*applet* queden reflectits en els diagrames que es van construint en els eixos de coordenades de la part inferior de l'*applet*. Per a veure l'evolució dels diagrames que representen el flux magnètic i la força electromotriu la figura 3.5 mostra l'equivalència entre la posició de l'espira i els diagrames:

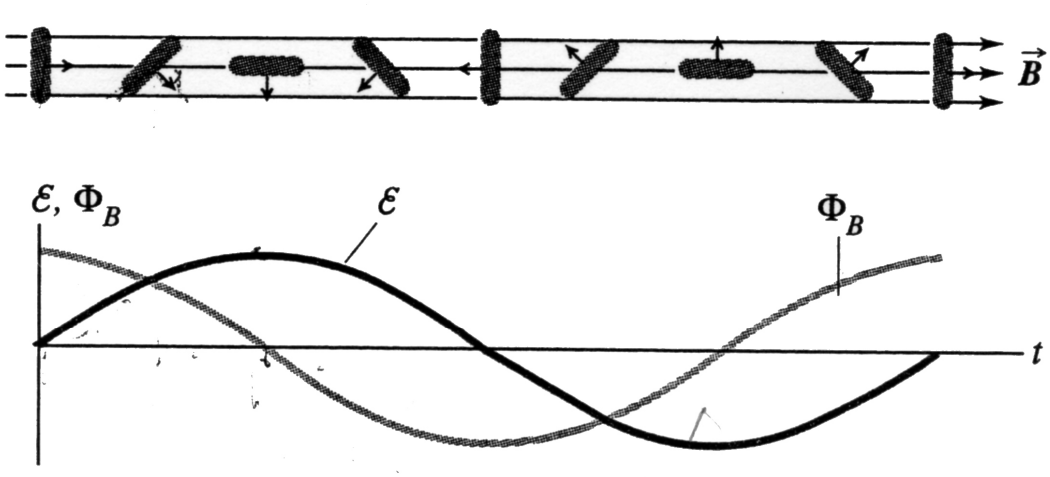

Figura 3.5. Correspondència de la posició de l'espira amb les gràfiques

- Els paràmetres interactius que permet l'*applet* són la velocitat angular (w) de l'espira i la intensitat del camp magnètic, són les eines que pot emprar l'usuari per a modificar l'*applet* i comprendre com afecten a l'amplitud i la freqüència de la tensió induïda.

3.3.2. Disseny i implementació

Aquest *applet* s'ha dissenyat en el paradigma de Programació Orientat a Objectes i mitjançant les directrius d'un programa de Java3D.

Primerament s'han realitzat uns quants esborranys de com hauria d'ésser l'*applet* i quins elements hauria de tenir, l'espai que hauria d'ocupar d'una *web*. El disseny visual ha quedat com a la figura 3.6.

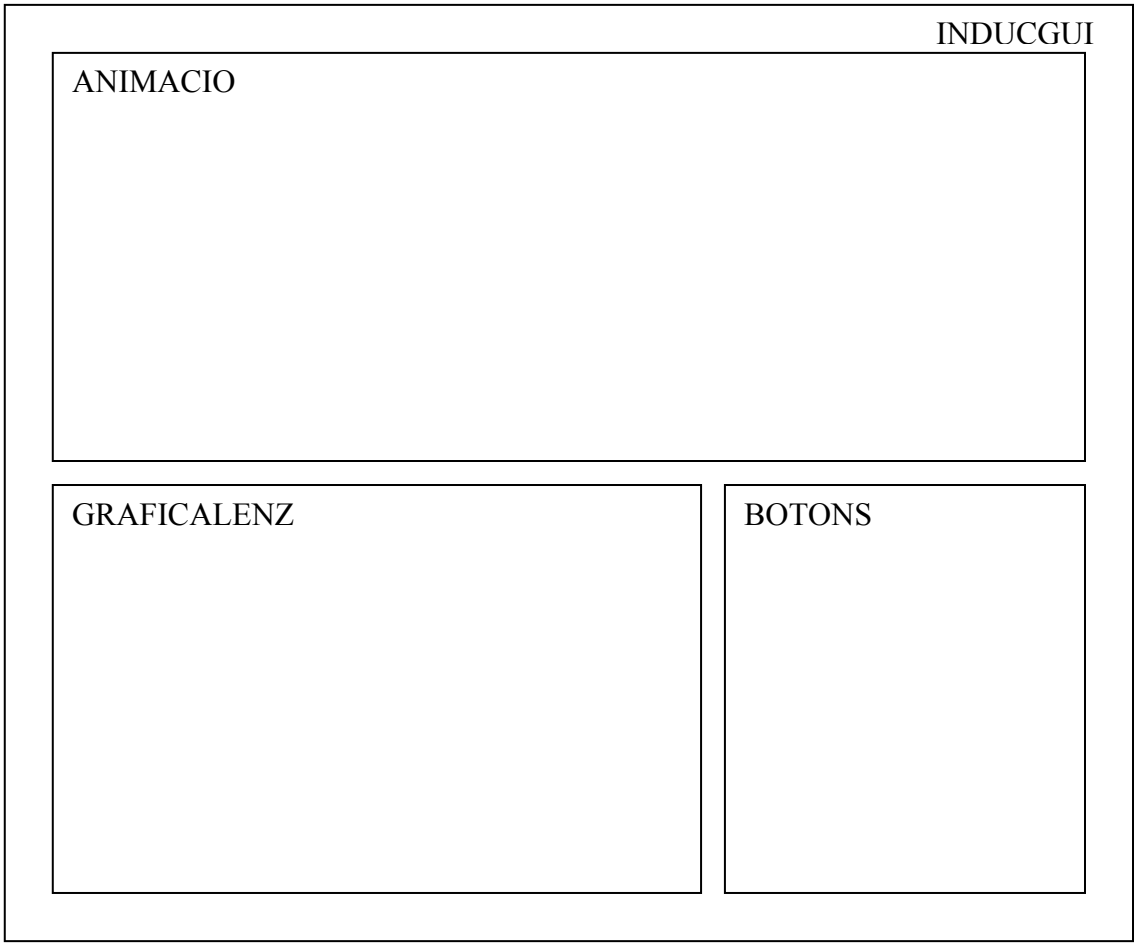

Figura 3.6. Disseny de la interfície de l'Applet 2

S'han definit els elements que compondrien l'*applet* i s'ha creat un diagrama de classes (Vegeu figura 3.7.).

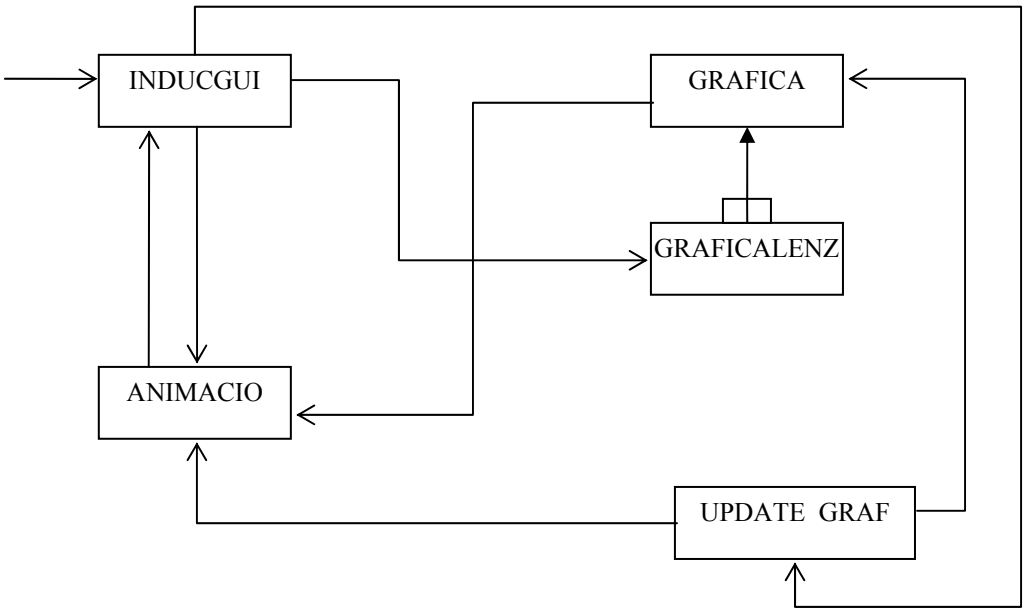

Figura 3.7. Diagrama de classes de l'Applet 2

INDUCGUI

S'ha determinat que la classe anomenada *InducGui* seria la que interaccionaria amb l'usuari i faria d'intermediari amb les altres classes. També s'ha decidit que també hauria de fer de contenidor d'aquestes altres classes ja que *InducGui* havia de ser *JApplet*.
En el seu constructor es creen i s'inicialitzen tots els elements (botons, animació i gràfica). A més a més s'insereixen aquests elements pel panell de contingut (*ContentPane*) segons la distribució *GridBagLayout* i la classe dels seus paràmetres *GridBagConstraints* de la biblioteca principal de Java 2.
A destacar d'aquesta classe és la utilització d'un temporitzador per a crear una classe anomenada *Updategraf*.

ANIMACIO

La classe *Animacio*, subclasse de *JPanel*, es divideix en dos *Canvas3D* que són el suport d'una animació de Java3D. En el seu constructor és on s'inicialitzen les animacions (mètode *inicialitzar*). Les animacions es creen en els mètodes *createSceneGraph* i *createSceneGraph2*.
A destacar d'aquests mètodes són les següents qüestions:

- La instrucció *simpleU.getViewingPlatform().getViewPlatformTransform()* és una crida a l'objecte *SimpleUnivers* que determina l'entorn on es genera el món en 3D, l'objecte que retorna és un *TransformGroup* des del que es pot configurar, mitjançant transformacions (*Transform3D*), el punt de vista de la realitat virtual creada.

- La utilització de quaternions (*Quat4f*) en les animacions generades per les subclasses de *javax.media.j3d.PathInterpolator* simplifica molt la definició de transformacions. Ja que només indicant les posicions intermèdies mitjançant classes tipus *Point*, les orientacions intermèdies mitjançant la classe *Quat4f* a més a més de definir un pes (*knot*) per a cada etapa. Aquesta manera de definir rotacions és una gran simplificació del que d'altra forma s'hauria de fer mitjançant un mètode molt més complicat com és la resolució de les equacions de Euler.

La creació de les rèpliques difuminades de la segona animació es fa en el mètode *fantasmes*. Es crea una de les rèpliques cada 45º (mitjançant el càlcul personalitzat del *Transform3D* a partir de l'interpolador) de color gris, exceptuant les de 90º, 180º i 270º que seran de color groc, verd i vermell respectivament, per a que coincideixin amb els colors dels punts de la gràfica. Aquesta diferenciació de les rèpliques, juntament amb la dificultat d'obtenir l'angle fa que el mètode *updat(Grafica grafic)* sigui complexe.

- El mètode d'obtenir l'angle de l'espira (de l'animació 1) es pot veure en *obtenirangle(float val)*. En primer lloc amb el valor d'*Alpha* s'obté, a partir de l'interpolador, el *Transform3D* equivalent. Amb aquest se'n extreu una matriu (*Matrix4f*) que defineix la rotació que aplicada a un objecte *AxisAngle4f* se'n obté l'angle en radians. Però aquest angle no és el que s'aprecia visualment donat que l'interpolador compta des de l'horitzontal i només fins 180º. Llavors per obtenir l'angle que volem (a partir de la vertical) s'han diferenciat tres casos:
    - Alpha<0.25: restem 90º a l'angle obtingut
    - Alpha>0.25 i Alpha<0.75: restem l'angle obtingut a 270º
    - Alpha>0.75: sumem l'angle obtingut a 270º

L'aparició i desaparició de les rèpliques es troba en el mètode *updat(Grafica grafic)*. Per a saber quins han de desaparèixer i quins aparèixer es compara l'angle per a saber en quina porció de 45º i assignar-li un número (el mateix que la rèplica, de 0 a 7). Els bucles que hi ha després són el que modifica l'aparença de les rèpliques. El primer bucle els mostra tots fins el número i el segon bucle amaga tots els del número+1 fins a 7. Seguidament es mostra el fragment del codi d'aquests bucles

```
...
for(int y=0;y<=i;y++){
        Group tot = (Group)replics.getChild(y);
        Group grupespira = (Group)tot.getChild(0);
        Shape3D cil = (Shape3D) grupespira.getChild(0);
        Appearance app = cil.getAppearance();
                TransparencyAttributes transAtt=new
          TransparencyAttributes(TransparencyAttributes.FASTEST,0.7f);
        app.setTransparencyAttributes(transAtt);
        cil.setAppearance(app);
        Group objfletxa = (Group) grupespira.getChild(1);
        Group objfletxa2 = (Group) objfletxa.getChild(0);
        Shape3D palfletxa = (Shape3D) objfletxa2.getChild(0);
        app = palfletxa.getAppearance();
        app.setTransparencyAttributes(transAtt);
```

```
            palfletxa.setAppearance(app);
        }
        for(int y=i+1;y<=7;y++){
            Group tot = (Group)replics.getChild(y);
            Group grupespira = (Group)tot.getChild(0);
            Shape3D cil = (Shape3D) grupespira.getChild(0);
            Appearance app = cil.getAppearance();
                    TransparencyAttributes transAtt=new
              TransparencyAttributes(TransparencyAttributes.FASTEST,1.0f);
            app.setTransparencyAttributes(transAtt);
            cil.setAppearance(app);
            Group objfletxa = (Group) grupespira.getChild(1);
            Group objfletxa2 = (Group) objfletxa.getChild(0);
            Shape3D palfletxa = (Shape3D) objfletxa2.getChild(0);
            app = palfletxa.getAppearance();
            app.setTransparencyAttributes(transAtt);
            palfletxa.setAppearance(app);
        }
...
```

En la primera part d'aquest mètode també és crida (en els moments necessaris) al mètode *Grafica.punts(int i, float valou)* que modifica quant se'l crida els punts de colors de 90º, 180º i 270º.

GRAFICA

Aquesta classe és una subclasse de *JPanel* i se li afegeix, mitjançant el Panell de Contingut, un objecte *Plot2DPanel* de la biblioteca JMathPlot.

GRAFICALENZ

Aquesta classe és una subclasse de *Grafica* i és qui s'encarrega d'actualitzar les dades de les gràfiques del *Plot2DPanel*. Hereta alguns mètodes, però els redefineix completament. Els dos mètodes més importants són el que actualitza les gràfiques, *updat()*, i el que destaca els punts especials (90º, 180º i 270º) anomenat *punts(int i, float valou)*.

- El primer d'aquests mètodes, *updat()*, serveix per actualitzar la gràfica. La primera part del condicional serveix per esborrar les dades i per inicialitzar les gràfiques amb el valor de 0. La segona part del condicional és qui actualitza les gràfiques després d'inicialitzar-les i durant una volta sencera. Les dades de cada volta sencera es van guardant en uns objectes *ArrayList* de l'Api principal de Java 2. Degut a que *JMathPlot* treballa amb arrays de *double* cada vegada que volem actualitzar les gràfiques hem de recórrer les dades per passar-les a arrays de *double* i així poder canviar les dades de les gràfiques.
- El mètode *punts(int i, float valou)* serveix per posar els punts de colors en el seu lloc. Realment els punts ja estan creats, però estan fora de l'àrea de visualització de les gràfiques i en aquest mètode el que es fa és canvia'ls-hi les dades ja que són diagrames de punts (scatterplot) de la biblioteca *JMathPlot*.

UPDATEGRAF

Aquesta classe és una subclasse de *TimerTask* de la biblioteca principal de Java2 (Vegeu [13]). Aquests tipus de classes serveixen per a executar una sèrie d'accions cada determinat temps. Sobrecarregant el seu mètode *run()* i posant l'objecte en un temporitzador s'aconsegueix que cada cert temps (determinat en el temporitzador, utilitzant la classe *Timer* de Java2) s'executi el mètode *run()*. En aquest cas s'utilitza per a actualitzar els diagrames i per capturar l'angle.

DIAGRAMES D'ESCENA

Els següents diagrames d'escena mostren l'estructura de les dues animacions de la que consta l'*Applet*.

Primera animació

El primer diagrama (Vegeu figura 3.8.) és de l'animació principal i mostra que per a fer la Rama de Representació s'ha utilitzat l'opció per defecte de *SimpleUnivers*. La Rama de Contingut comença per *objRoot* (*BranchGroup*) del que en surten dues branques filles: una per posar el fons d'un color (*fondo*) i *objRoot2* per seguir amb la Rama de Contingut.
objRoot2 es divideix en quatre branques en les que separar el tipus d'elements de l'animació.

- *objStatic*: branca que engloba tots els objectes que no es mouen.
- *objFletxes*: branca que agrupa totes les fletxes de camp.
- *objMov*: branca de la que penja l'espira que rota.
- *DirectionalLight*: focus de llum per il·luminar l'escena.

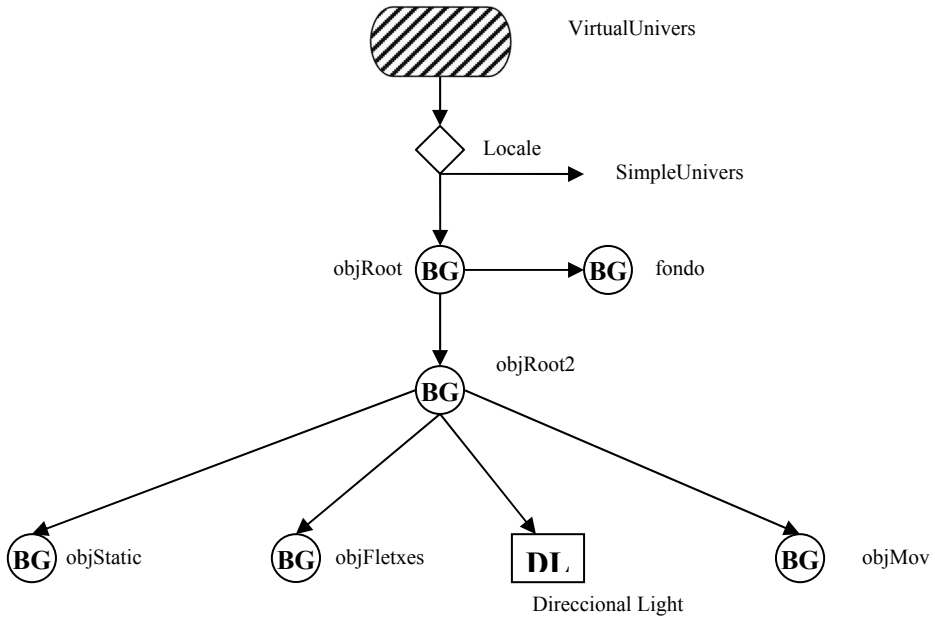

Figura 3.8. Diagrama de la primera animació

objStatic

Aquesta branca (Vegeu figura 3.9.) engloba, com es pot veure en el diagrama següent, les parts no movibles de l'animació. Les 9 branques amb que es divideix són *TransformGroup* que serveixen per posicionar els objectes que pengen d'ells que són, d'esquerra a dreta:

- *ColorCube*: utilitzats per representar els pols de l'imant.
- *Cylinder*.BODY: part d'un cilindre, utilitzats per simular l'anella de contacte amb l'espira que rota.
- Les lletres N, S i V que representen, respectivament, els pols de l'imant i el voltímetre que mesura el potencial induït en l'espira.
- Cable1 i cable2 simulen els cables de contacte entre les anelles i el voltímetre.

objFletxes

Aquesta branca (Vegeu figura 3.10.) reuneix totes les estructures en forma de fletxa que simbolitzen la intensitat del camp magnètic que es produeix entre els dos pols de l'imant.
L'estructura que forma una fletxa es compon d'un *BranchGroup* per a agrupar-ho i poder-ho manejar millor. El *TransformGroup* que el segueix serveix per posicionar tota l'estructura. El segueix un objecte *Shape3D* i dos *TransformGroup*. El primer és una línia que representa el pal de la fletxa, els dos *TransformGroup* posicionen uns petits cons utilitzats com a punta de fletxa.
Aquesta estructura, en un principi, està repetida 4 vegades i representa el camp mínim. Però depenent de quin camp se seleccioni l'estructura estarà repetida 8 vegades, pel camp doble, o 12 vegades, pel camp triple.

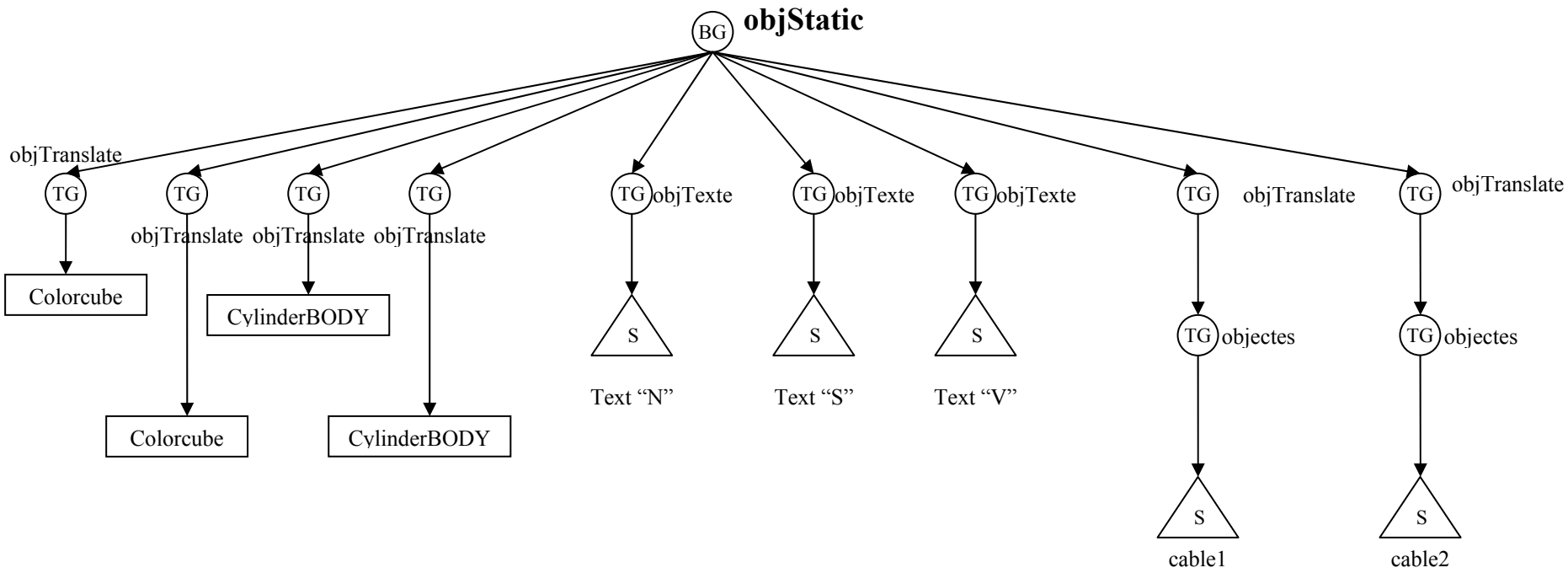

Figura 3.9. Branca *objStatic*

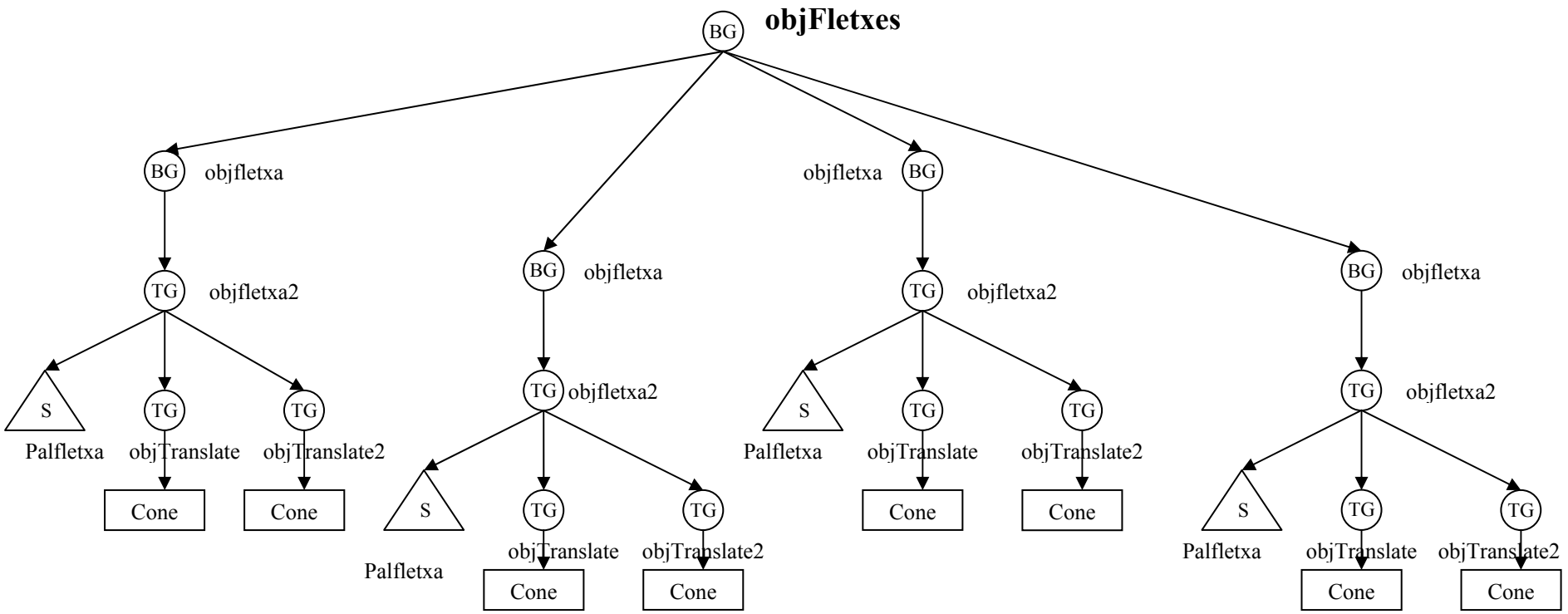

Figura 3.10. Branca *objFletxes*

objMov

Per últim la branca (Vegeu figura 3.11.) de *objMov* compren la part que es mou de l'animació, que és l'espira. Es compon d'un *BranchGroup*(*objMov*) per al seu manegament posterior; de dos *TransformGroup*, *objtranslate3* per a definir la posició, *objspin* per a definir la rotació. L'interpolador *rotator* (simbolitzat per la B de *Behavior*) penja de *objtranslate3* a l'igual que *objspin* i al mateix temps s'aplica sobre *objspin*. I finalment l'objecte *cableespira*(*Shape3D*) que és l'espira.

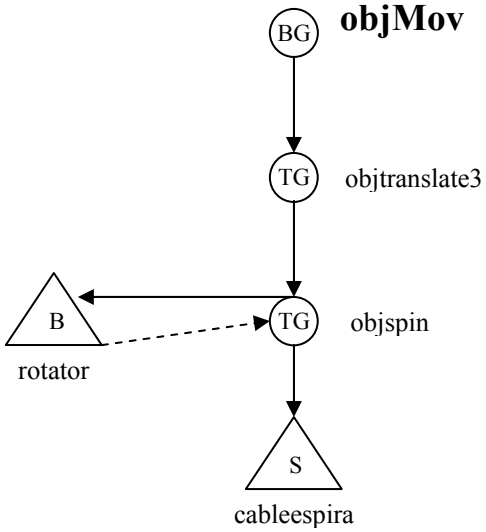

Figura 3.11. Branca *objMov*

Segona animació

En aquesta segona animació (Vegeu figura 3.12.), la Rama de Representació és igual a la de la primera animació, és a dir, utilitza la versió predeterminada que és l'ús de *SimpleUnivers*. Després d'afegir un fons a l'animació la Rama de Contingut es divideix en varies branques:

- *fletxes*: Tres fletxes amb la mateixa estructura de la primera animació.
- *lletra*: la lletra "B" amb la fletxa a sobre que simbolitza el camp magnètic.
- *transespira*: el cilindre simbolitza l'espira de la primera animació vista de perfil amb una fletxa que representa la Normal. Juntament amb l'interpolador que fa moure la representació de l'espira.
- *Replics*: és una estructura que engloba un conjunt de rèpliques de l'objecte anterior (el cilindre i la fletxa). Aquestes estan posicionades cada 45º i van apareixent quan l'objecte animat els passa per sobre. L'estructura presentada al diagrama d'escena és només una de les rèpliques, totes les altres (8 en total) són filles de *Replics*.

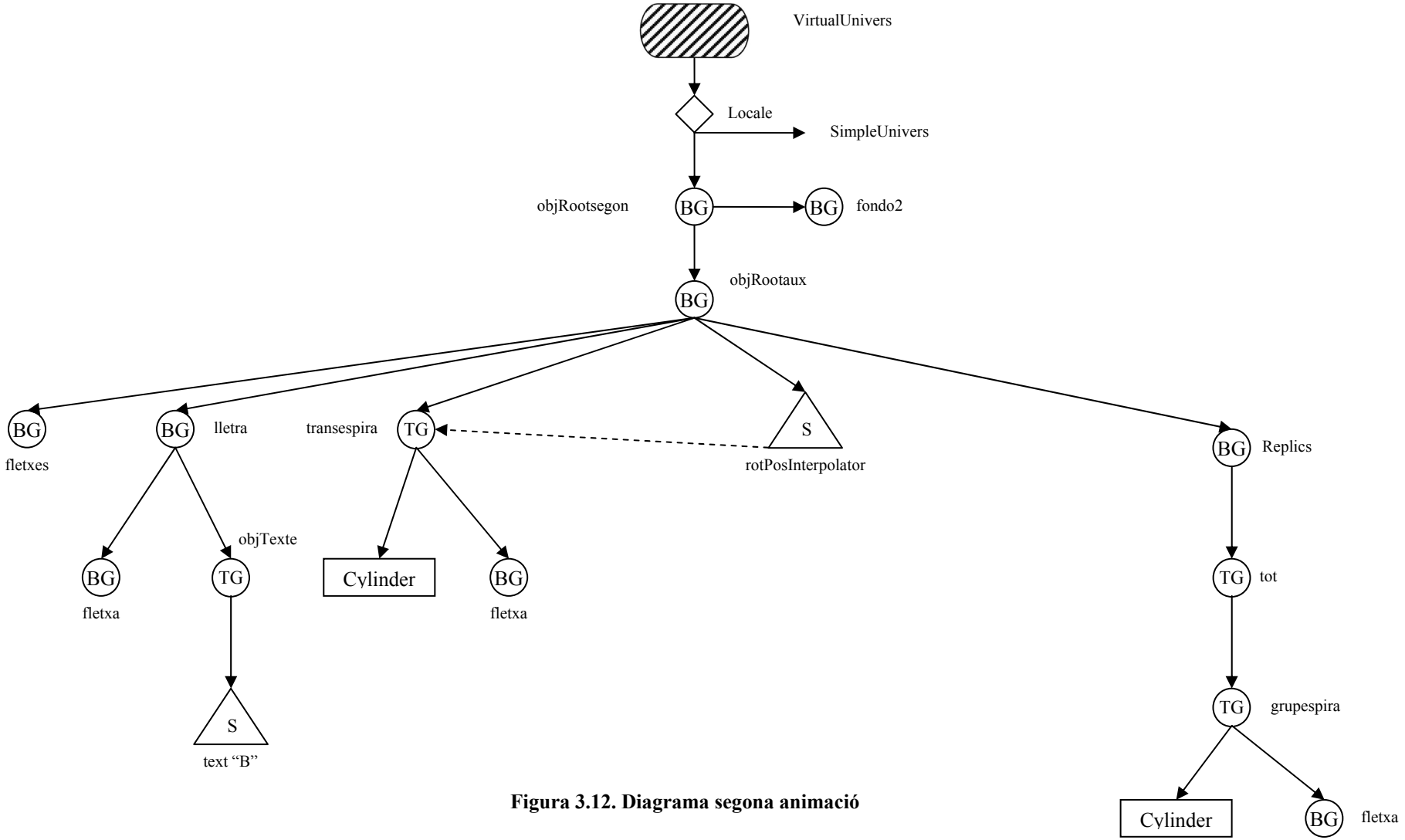

Figura 3.12. Diagrama segona animació

3.3.3. Instal·lació i ús

Per a poder utilitzar l'aplicació cal seguir les següents directrius:

- Utilitzar un navegador amb el *plugin* de Java (Vegeu instal·lat, la versió del qual ha de ser com a mínim la 1.5.0.
- Tenir instal·lada la biblioteca Java 3D 1.5.0.
- Si s'utilitza el fitxer *html* que acompanya l'arxiu *JAR* cal que tots dos estiguin en el mateix directori.
- Si es vol utilitzar des d'un fitxer *html* propi cal cridar l'aplicació amb la següent clàusula:

```
<APPLET CODEBASE="." ARCHIVE="TFC.jar" CODE="tfc.InducGUI.class" WIDTH="600" HEIGHT="600">
  </APPLET>
```

On en el paràmetre *CODEBASE* cal posar-hi la ruta fins a l'arxiu TFC.jar, a partir de l'arxiu *html*.

L'ús de l'aplicació és bastant senzill. Consta de 3 botons i dues llistes *ComboBox*) seleccionables.

- El botó *START* serveix per engegar o retornar, després d'una pausa, l'animació.
- El botó *PAUSE* per aturar l'animació.
- El botó *STEP* serveix per fer moure l'animació pas a pas.
- La llista "Intensitat de camp" serveix per seleccionar la intensitat del camp magnètic. Els valors que pot prendre són B, 2B i 3B.
- La llista "Velocitat angular " indica la velocitat de rotació de l'espira i els seus valors poden ser W, 2W i 3W.

La utilització de la intensitat de camp magnètic fa que l'amplitud de l'ona sigui major, per tant es generi una major variació de flux magnètic i el qual produeix una major força electromotriu. (Vegeu figura 3.13 i 3.14)

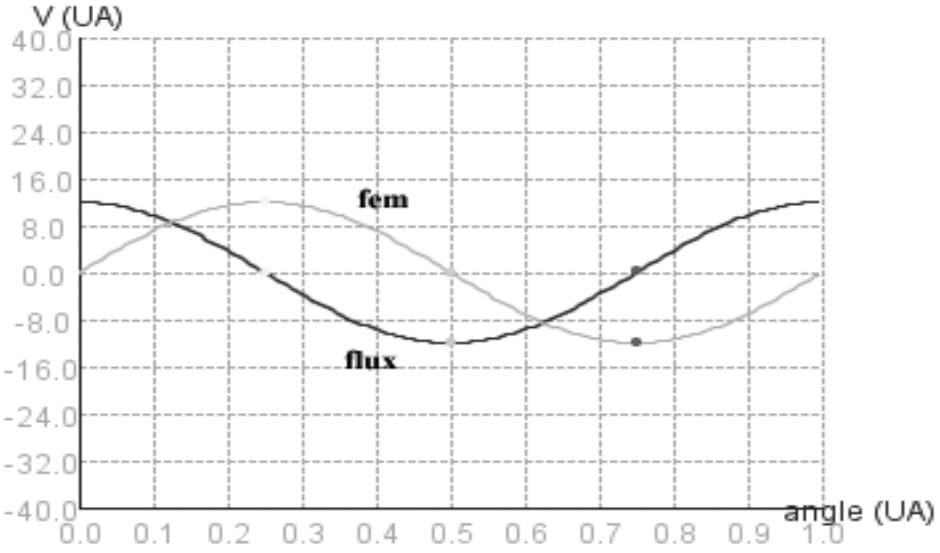

Figura 3.13. Gràfiques a velocitat 2w i intensitat 3B

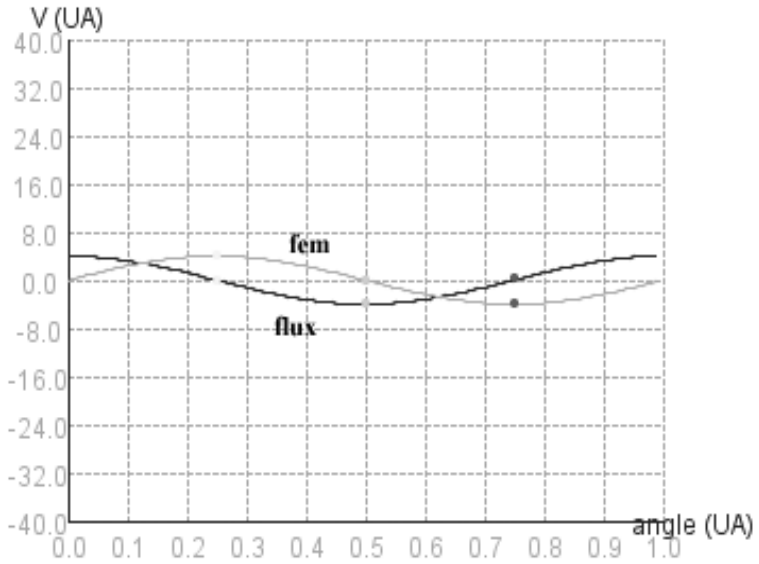

Figura 3.14. Gràfiques a velocitat 2w i intensitat B

La modificació del paràmetre de la velocitat angular provoca que la freqüència sigui molt més gran i per tant la variació de temps molt més petita, a la vegada que una major variació de flux. Això segons la fórmula de la Llei de Faraday (Vegeu [3], [4] i [5]) provoca una major força electromotriu ja que es produeix més variació de flux en menys temps. (Vegeu figura 3.15)

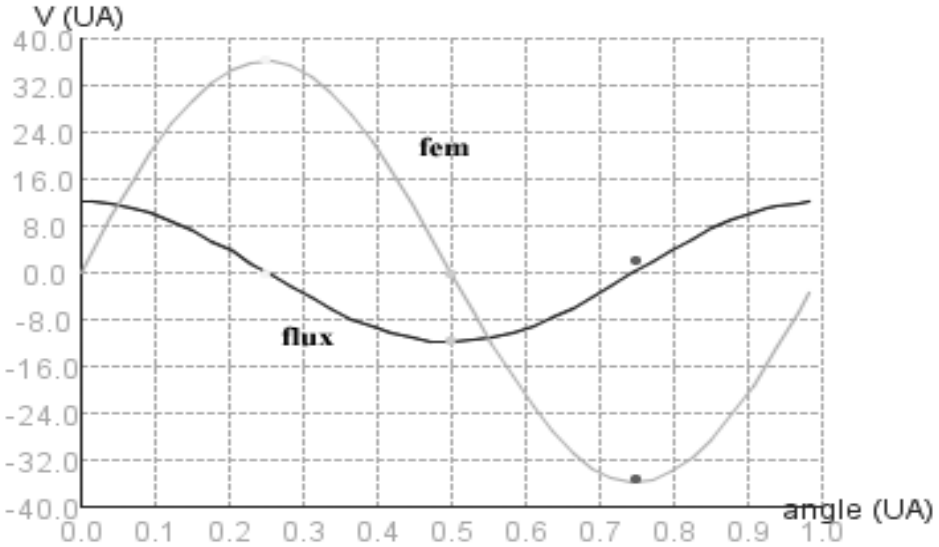

Figura 3.15. Gràfiques a velocitat 3w i intensitat 3B

4. Conclusions i treball futur

La realització d'aquest projecte m'ha ajudat a adquirir uns coneixements molt importants per al meu desenvolupament com a professional. Un aprenentatge i una porta d'accés al món laboral.

Després d'explorar les dues alternatives que vaig provar per a dissenyar *applets* relacionats amb la física, en la que més he pogut crear sense estar limitat per uns mètodes predefinits ha estat amb l'API de Java3D, mentre que les possibilitats amb els *Physlets* són molt escasses. Tenint en compte que l'objectiu del projecte era explorar les possibilitats d'ambdues biblioteques *Physlets* i Java3D, sospesant les virtuts de cadascuna qualitat matemàtica, per part dels *Physlets*, qualitat visual, per part de Java3D; he arribat a la conclusió que la millor opció és la utilització de la biblioteca Java3D per la gran capacitat visual que té i per les amplies capacitats matemàtiques que proporciona la biblioteca principal i la pròpia Java3D. (Vegeu [6],[7],[8],[9] i [10])

Després de la conclusió d'aquest projecte i veient els resultats estic molt satisfet del que s'ha aconseguit, sobretot amb el *applet* dissenyat en Java3D, i crec que els objectius de crear unes aplicacions per a un ús pedagògic s'ha acomplert.

Malgrat el finiment d'aquest projecte m'agradaria proposar alguna observació que crec que en projectes futurs seria útil, com ara aprofundir en l'estudi de la classe *Alpha* i modificar-la (subclasse) per a crear-ne una que s'adaptés més a les nostres necessitats. L'ús de fils d'execució seria molt útil alhora d'una major fluïdesa en animacions més complexes. A més a més de l'ús o creació d'una biblioteca diferent a *JMathTools*, com ara *JFreeChart*, més depurada. Un tema que també seria molt útil aprofundir seria l'estudi i la comprensió més profunda dels quaternions, eina vital en la utilització de transformacions en Java3D.

Apèndix A: Disseny d'una pàgina web de suport als applets

Donat que l'objectiu final dels *applets* que s'han presentat són per a utilitzar-los en una pàgina *web* amb finalitat pedagògica dels Fonaments Físics de la Informàtica, s'ha dissenyat un prototip, un esquelet, del que podria ser aquesta *web* amb un índex dels temaris que hi apareixerien.
Per a dissenyar aquesta *web* s'ha utilitzat l'editor *FrontPage* 2003.
La pàgina inicial s'ha anomenat *index.html* i està dividida en tres marcs, dels quals dos són fixos, *menu.html* i *titol.html*, i l'altre és variable depenent de l'apartat del menú en el que s'entri.
L'aparença d'aquesta *web* (Vegeu figura A.1), mostrant un dels *applets* dissenyats, és aquesta:

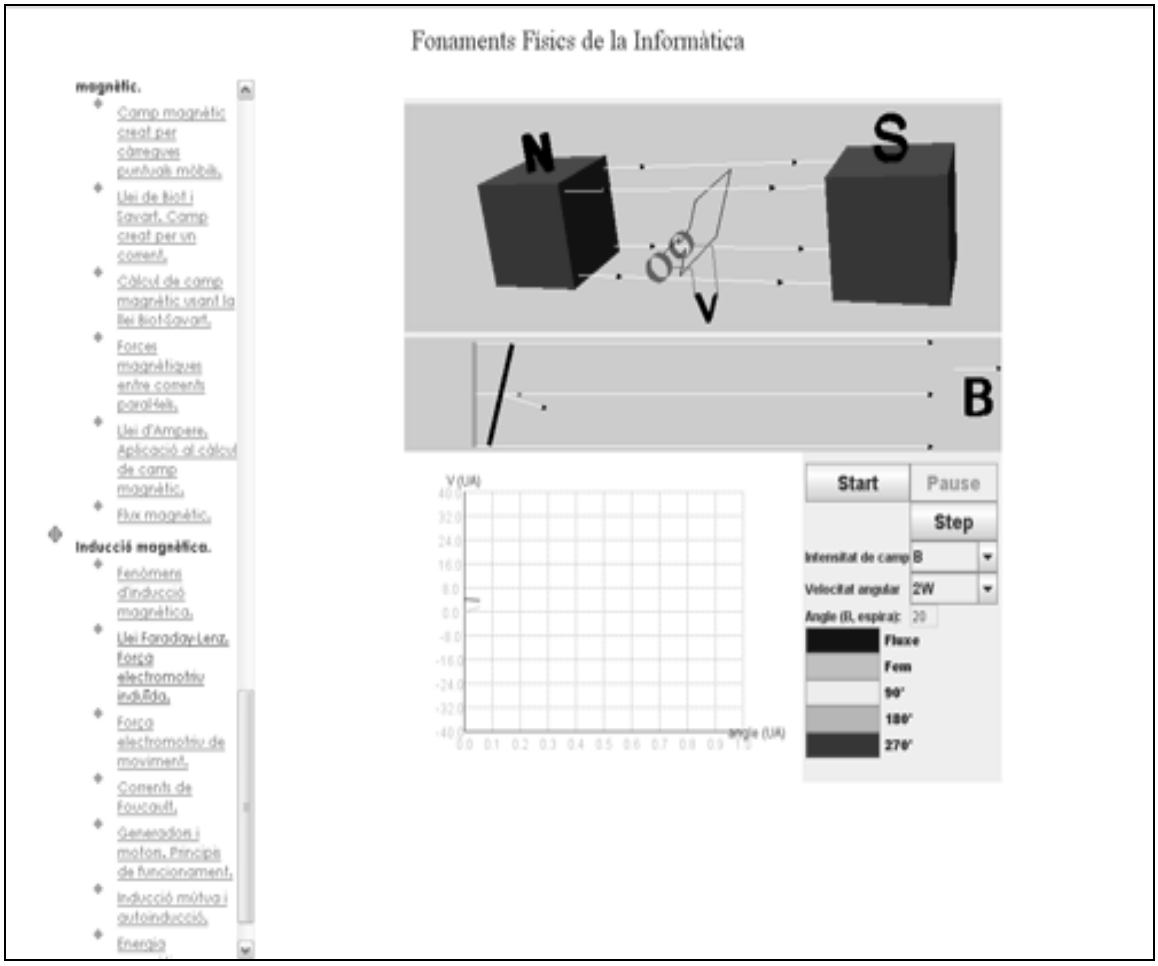

Figura A.1. Aparença de la *web* amb un dels *applets*

El menú de la *web* i les seves corresponents pàgines *web* és aquest:

Camp elèctric.
- Càrrega elèctrica. Llei de Coulomb.
- Camp elèctric.
- Càlcul de camp elèctric mitjançant la llei de Coulomb.
- Flux del camp elèctric. Llei de Gauss.
- Càlcul del camp elèctric mitjançant la llei de Gauss.

- Moviment d'una partícula carregada en un camp uniforme. L'oscil·loscopi.
- Conductors en equilibri electrostàtic. Càrrega i camp.

Potencial elèctric.
- Energia potencial electrostàtica i potencial elèctric.
- Potencial en un sistema de càrregues puntuals.
- Potencial en distribucions contínues de càrrega.
- Relació general entre camp elèctric i potencial.
- Superfícies equipotencials.

Condensadors. Dielèctrics.
- Condensadors. Capacitat.
- Energia elèctrica emmagatzemada en un condensador.
- Densitat d'energia d'un camp electrostàtic.
- Dielèctrics. Polarització.
- Condensadors amb dielèctrics.

Electrocinètica.
- Corrent elèctric. Densitat de corrent.
- Llei d'Ohm. Resistència elèctrica.
- Resistivitat.
- Conducció elèctrica en conductors i semiconductors.

Camp magnètic.
- Definició i propietats del camp magnètic. Força magnètica.
- Força magnètica sobre una càrrega mòbil.
- Força magnètica sobre un element de corrent i un conductor.
- Imants a l'interior d'un camp magnètic. Moment magnètic.
- Acció d'un camp magnètic uniforme sobre una espira.
- Moviment de càrregues a l'interior d'un camp magnètic. Aplicacions.
- Efecte Hall. Sensors de camp magnètic.

Fonts del camp magnètic.
- Camp magnètic creat per càrregues puntuals mòbils.
- Llei de Biot i Savart. Camp creat per un corrent.
- Càlcul de camp magnètic usant la llei Biot-Savart.
- Forces magnètiques entre corrents paral·lels.
- Llei d'Ampere. Aplicació al càlcul de camp magnètic.
- Flux magnètic.

Inducció magnètica.
- Fenòmens d'inducció magnètica.
- Llei Faraday-Lenz. Força electromotriu induïda.
- Força electromotriu de moviment.
- Corrents de Foucault.
- Generadors i motors. Principis de funcionament.
- Inducció mútua i autoinducció.
- Energia magnètica.

Apèndix B: Biblioteques i software matemàtic i de visualitzacions gràfiques

- *JFreeChart*: És una biblioteca de diagrames programat en Java, similar a *JMathTools* (Vegeu [11]), i està dissenyada com a complement per a una aplicació per dibuixar diagrames de diversos tipus d'alta qualitat. La versió actual és la 1.0.6. (Vegeu [12]) Es caracteritza per:
    - Una *API* coherent i ben documentada que suporta gran quantitat de diagrames.
    - Un disseny entenedor i fàcilment subclassificable. Tant en aplicacions de servidor com de client.
    - Permet extreure les dades en gran quantitat de formats tant per utilitzar dins d'una aplicació Java (*Swing*) o per aplicacions externes, com ara imatges (jpg, png, etc.) o d'altres com en pdf.
    - Està sota llicència LGPL (Gnu Lesser General Public License), per tant és programari lliure, és a dir pot ser copiat, distribuït i/o modificat, però al mateix temps pot formar part d'un programari propietari.

  Aquesta biblioteca té 7 anys d'existència i són molts els productes o projectes que la utilitzen, per posar alguns exemples tant de programari lliure com propietari, és utilitzada per *Google* en el producte *Google Adwords*, en la *web* de *NetBeans* o en el projecte *Eclipse* per a presentar estadístiques o per a fer càlculs.

Bibliografia

[1] **Fislets. Enseñanza de la Física con Material Interactivo**
Ed. PEARSON EDUCACIÓN S.A, Madrid 2004

[2] **Davidson Physics**
http://www.phy.davidson.edu/
http://webphysics.davidson.edu/

[3] **Física para la ciencia y la tecnología**
Paul A. Tipler, Editorial Reverté 4ª edición

[4] **Física**
Sears / Zemansky, Editorial Aguilar 1ª edición 6ª reimpresión

[5] **Wikipedia**
http://www.wikipedia.org/

[6] **Programación en 3D con Java3D**
J.J. Pratdepadua, Editorial RA-MA 2003

[7] **Sun's Java3D Homepage**
http://java.sun.com/products/java-media/3D/

[8] **Java3D Project Home**
https://java3d.dev.java.net/

[9] **Java 3D - 3D Graphics Tutorial and Information**
http://www.java3d.org/

[10] **The Java 3D Graphics Community**
http://www.j3d.org/ (i subwebs)

[11] **JMathTools**
http://jmathtools.sourceforge.net

[12] **JFreeChart**
http://www.jfree.org/jfreechart/index.html,
http://sourceforge.net/projects/jfreechart

[13] **Sun Microsystems**
http://www.sun.com